Titre: Title:	Commande collaborative d'un fauteuil roulant dans un environnement partiellement connu
Auteur: Author:	Dang Quang Nguyen
Date:	2017
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Nguyen, D. Q. (2017). Commande collaborative d'un fauteuil roulant dans un environnement partiellement connu [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/2929/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/2929/
Directeurs de recherche: Advisors:	Richard Gourdeau, & Joëlle Pineau
Programme: Program:	génie électrique

UNIVERSITÉ DE MONTRÉAL

COMMANDE COLLABORATIVE D'UN FAUTEUIL ROULANT DANS UN ENVIRONNEMENT PARTIELLEMENT CONNU

DANG QUANG NGUYEN
DÉPARTEMENT DE GÉNIE ÉLECTRIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE ÉLECTRIQUE)
DÉCEMBRE 2017

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

COMMANDE COLLABORATIVE D'UN FAUTEUIL ROULANT DANS UN ENVIRONNEMENT PARTIELLEMENT CONNU

présenté par : NGUYEN Dang Quang
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de :

M. SAUSSIÉ David, Ph. D., président
M. GOURDEAU Richard, Ph. D., membre et directeur de recherche
Mme PINEAU Joëlle, Ph. D., membre et codirectrice de recherche
M. LE NY Jérôme, Ph. D., membre

DÉDICACE

À tous mes amis du labos,
vous me manquerez...

REMERCIEMENTS

C'est un grand plaisir pour moi de remercier toutes les personnes qui ont permis à ce travail d'être ce qu'il est.

Je remercie énormément mes directeurs de recherche, Richard Gourdeau et Joelle Pineau pour leur soutien, leur encadrement et le projet intéressant qu'ils m'ont proposé.

Je remercie le conseil de recherches en sciences naturelles et en génie du Canada (CRSNG) pour son aide financière pendant mes recherches.

Je remercie tous les professeurs qui m'ont enseigné à l'École Polytechnique de Montréal et à l'Institut National des Sciences Appliquées de Lyon en France. En particulier, je remercie les professeurs Jérôme Le Ny et David Saussié pour avoir accepté d'être membres du jury pour ce mémoire.

Je remercie les membres du laboratoire MRASL, en particulier André Phu-Van Nguyen et Duc-Tien Nguyen avec qui j'ai passé le plus de temps au laboratoire. J'ai partagé avec eux des moments inoubliables et ils étaient toujours présents pour m'épauler et me divertir. Nos discussions étaient ma source d'inspiration et d'encouragement.

Je remercie tous mes amis que j'ai rencontrés pendant mon séjour à Montréal. Je tiens à remercier également tous mes amis en France et au Vietnam qui me manquent beaucoup.

Enfin et surtout, j'exprime ma plus profonde gratitude à mes parents, ma sœur et mes amis, pour leurs encouragements et leur soutien à l'épreuve du temps et de la distance qui nous sépare. Sans leur amour et leur bienveillance, je n'aurais jamais pu réaliser tout ce que j'ai réalisé.

RÉSUMÉ

Depuis des années, des chercheurs et des étudiants aux cycles supérieurs de trois universités au Québec ont travaillé au projet de réalisation d'un prototype d'un Fauteuil Roulant Motorisé Intelligent (FRMI). C'est un robot mobile équipé de capteurs, de caméras et de modules de contrôle permettant à la chaise roulante d'accomplir de nombreuses tâches autonomes avec assistance aux utilisateurs.

Malgré le bon état d'avancement de ces travaux de recherche, il nous reste plusieurs aspects à améliorer et perfectionner. En effet, le fauteuil possède des modules de cartographie et localisation simultanées (Simultaneous Localization And Mapping (SLAM)), de navigation complètement autonome, et de contrôle collaboratif qui assiste les usagers en combinant les commandes de téléopération et autonomes. Pour utiliser le contrôle collaboratif, il faut déterminer une ou plusieurs destinations potentielles dans la carte de l'environnement. Avec les commandes données par l'utilisateur, le système estime son intention et l'aide à atteindre la destination. Pour les personnes âgées, les utilisateurs principaux des fauteuils roulants, il n'est pas concevable de naviguer pour créer la carte de l'environnement, de la lire et d'y placer des destinations potentielles.

Pour simplifier la tâche de l'utilisateur, nous devons localiser le véhicule dans une carte connue *a priori* et utiliser cette carte dans un algorithme de SLAM et inférer automatiquement l'intention de l'utilisateur au cours de la navigation. Le but de ce projet est alors de compléter l'environnement de travail autour du module de contrôle collaboratif en améliorant le module de cartographie et de créer un module qui détermine des destinations potentielles dans la carte afin de les placer automatiquement dans l'environnement de navigation de la chaise.

La réalisation de ces objectifs est accomplie par une étude du module de contrôle collaboratif, l'insertion de la carte connue *a priori* dans un algorithme de SLAM et la détection automatique de points d'intérêt dans cette carte. La première partie consiste à comparer des algorithmes de SLAM existants pour choisir le plus approprié à notre application. D'un autre côté, une technique de construction des données de SLAM à partir d'une carte de l'environnement est implémentée et testée grâce à un algorithme qui crée une carte de type SLAM à partir d'une génération de trajectoires virtuelles. La deuxième partie consiste à détecter les destinations potentielles dans la carte en cherchant les salles et des points d'intérêts

intermédiaires dans les couloirs.

Notre travail contribue à simplifier l'utilisation de la chaise et à améliorer l'expérience des usagers face à ces nouvelles technologies. Il permet de rendre l'algorithme de contrôle collaboratif facilement utilisable par un plus grand nombre d'utilisateurs en assurant que les destinations potentielles ne sont pas placées de façon arbitraire par les usagers. Le programme a subi plusieurs tests en simulation et avec le système réel, et les résultats montrent que, dans divers scénarios, notre système a bien détecté l'intention de l'utilisateur et s'est rendu à la bonne destination.

ABSTRACT

Researchers and students in three universities of Quebec, École Polytechnique de Montréal, University de Montréal and McGill University have been working on the Intelligent Powered Wheelchair (IPW) for many years. A wheelchair which has the ability to do some daily tasks with the help of sensors and computers was developed. It can navigate autonomously or create a map of the environment by itself. Therefore, it can be considered as a mobile robot.

Despite the fact that the IPW is in a very advanced state and was developed for years, improvements can still be made. Indeed, the chair possessed a SLAM module to create a map and localize the chair in the environment, a navigation module which can navigate autonomously inside the map or assists the user to control the chair without ignoring their command by combining the autonomous navigation with the manual navigation. It is called shared autonomy. In order to use the shared autonomy, first of all, we need to determine the possible destination on the map. Then, the controller will analyse the manœuver of the user and after that, it will guess the user's intention before leading the robot to the destination. For the main user of the wheelchair, the elderly, the utilization of such function is not obvious.

In order to simplify the user's works, we need to locate the robot in the map and then define all the destinations automatically. The project completes the workflow of the shared autonomy control by improving the SLAM algorithm of the IPW and by creating a module which analyses the map and determines all the destinations.

In this thesis, we analyse many methods of SLAM available before deciding to implement one in our chair. Second, we will modify the design of the algorithm so we can load a map before navigating by creating a virtual trajectory which navigate the existing map. Finally, we create a module that can find all the destinations in the map and manage to put it in the SLAM all automatically.

We want to improve the user experience by simplifying its tasks. Our works can support the shared autonomy algorithm to reach as many patients as possible, even the one who are not used to modern technology. We tested our program in the simulation to assure the fluidity and the functionality of the system.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

FRE	Fauteuil Roulant Électrique
FRMI	Fauteuil Roulant Motorisé Intelligent
SLAM	Localisation et cartographie simultanées (Simultaneous Localization And Mapping)
LIDAR	Détection de la lumière et télémétrie (LIght Detection And Ranging)
IMU	Centrale inertielle (Inertial Measurement Unit)
GPS	Géo-positionnement par satellite (Global Positioning System)
ROS	Système d'exploitation du robot (Robot Operating System)
IHM	Interface Homme-Machine
KF	Filtre de Kalman (Kalman Filter)
EKF	Filtre de Kalman étendu (Extended Kalman Filter)
SIR	*Sampling Importance Resampling*
SPA	*Sparse Pose Adjustment*
LM	Levenberg-Marquardt
GN	Gauss-Newton
CCS	*Compressed Column Storage*
DFS	Algorithme de parcours en profondeur (Depth First Search)
RVIZ	Programme de visualisation de ROS (Ros VIsualiZation)
ICP	*Iterative Closest Point*
HMM	Modèle de Markov caché (Hidden Markov Model)
SIFT	Transformation de caractéristiques visuelles invariante à l'échelle (Scale Invariant Feature Transform)
IPW	Fauteuil Roulant Intelligent (Intelligent Powered Wheelchair)
AMCL	Localisation adaptative de Monte Carlo (Adaptive Monte Carlo Localization)
KLD	Divergence de Kullback–Leibler
DXF	*Drawing eXchange Format*

LISTE DES ANNEXES

CHAPITRE 1 INTRODUCTION

Un bon pourcentage d'utilisateurs de Fauteuil Roulant Électrique (FRE) à mobilité réduite souffre de tremblements des mains ou de troubles musculaires limitant leur capacité à contrôler leur fauteuil avec le joystick. La mobilité réduite implique fréquemment une réduction de la sociabilité du patient, ce qui peut, sur le long terme, conduire à l'isolement, l'anxiété et la dépression. Pour améliorer la qualité de vie des utilisateurs de FRE incapables d'utiliser le joystick, des modules de contrôle totalement autonomes sont développés afin de réaliser des missions de navigation dans les environnements inconnus (LURCH, SmartWheeler, NavChair, FRMI, etc.). Par contre, les êtres humains sont habituellement plus plurivalents et sont capables de mieux manipuler les tâches complexes. Il est alors intéressant de ne pas négliger l'une ou l'autre manière de contrôler (manuelle et automatique) mais de combiner les deux façons afin de mieux exploiter les deux sources de données et de mécanismes de contrôles. C'est pourquoi nous avons concentré nos efforts sur le développement et le perfectionnement de la commande collaborative et ses activités auxiliaires pour garantir l'autonomie et le confort des utilisateurs du FRE.

1.1 Définitions et concepts de base

Le FRMI est un FRE équipé pour pouvoir bien assister les utilisateurs. Il possède un ordinateur de bord, un écran tactile avec divers capteurs : une centrale inertielle (Inertial Measurement Unit (IMU)), un GPS et des encodeurs magnétiques, des LIDAR et des caméras dont une caméra 3D Kinect. La figure 1.2 présente les deux prototypes réalisés, dont un pour la navigation dans les grands environnements extérieurs et l'autre pour naviguer à l'intérieur. Ces dispositifs nous permettent de réaliser plusieurs tâches complexes :

Localisation : Pour pouvoir bien fonctionner individuellement, la chaise doit bien se localiser dans l'environnement afin de trouver une trajectoire globale. En utilisant des modules IMU, GPS et les encodeurs, elle peut se localiser dans les environnements intérieurs et extérieurs.

Cartographie : La cartographie est une des tâches de base pour pouvoir planifier la trajectoire dans un environnement. On collecte les données fournies par les capteurs pour créer ou améliorer une carte de l'environnement. En testant plusieurs façons de cartographier, on peut déterminer une meilleure méthode pour chaque application (navigation dans les environnements intérieurs ou extérieurs).

(a) Chaise avec trois LIDAR à courte portée

(b) Chaise avec deux LIDAR à longue portée

Figure 1.2 Nos deux prototypes de la chaise roulante. Celle à droite est équipée des LIDAR à longue portée avec le GPS pour les application dans les environnements extérieurs. Celle à gauche est équipée des LIDAR de faible portée pour des tests dans les environnements intérieurs.

Navigation autonome : Afin de naviguer de façon autonome, la chaise doit compléter les deux tâches de navigation locale (suivi de mur, évitement d'obstacles, etc.) et de navigation globale (la localisation du robot et la planification de trajectoire avec la carte) tout en même temps.

Navigation collaborative : Afin de naviguer avec la commande collaborative, la chaise doit avoir assez de puissance de calcul pour pouvoir prévoir l'intention de l'utilisateur et la combiner avec la navigation autonome de façon adéquate en évitant les contradictions dans le contrôle. On essaie de réaliser le contrôle autonome entre le niveau deux et trois [1] suivant le standard J3016 de SAEInternational (2017), ce qui partage la tâche de superviser le déplacement dans l'environnement aux deux agents de commandes : l'utilisateur et le robot.

C'est ainsi un cas très particulier de robot mobile : le fauteuil peut naviguer en observant l'environnement et en réagissant aux événements de deux façons différentes, voire opposées (du point de vue de l'utilisateur et de celui du robot). Par exemple, les utilisateurs de FRE souffrent souvent des handicaps moteurs (tremblement, trouble musculaire, etc.) qui diminuent leurs réflexes. Par contre, leurs capacités cognitives restent intactes. Ils peuvent ainsi

1. Niveau deux : automation partielle : Seulement l'exécution du pilotage, de l'accélération et du ralentissement du robot est faite par le système.
Niveau trois : automation conditionnelle : Des modes spécifiques de la navigation autonome sont faits par le robot, l'usager doit surveiller le robot en tout temps.

prendre de meilleures décisions dans les situations complexes du point de vue de la navigation globale que la commande autonome qui va peut-être faire un arrêt dans ce cas. Par contre, dépendamment de l'aptitude et de la dextérité de l'utilisateur, la navigation dans les environnements avec obstacles nombreux peut induire des situations dangereuses. D'où la nécessité du contrôle partagé, spécialement pour ce robot mobile - la chaise roulante.

Tous les modules du FRMI sont implémentés dans l'environnement Robot Operating System (ROS). ROS est un méta-système d'exploitation qui est conçu pour les applications robotiques complexes. Son architecture modulaire permet de choisir et d'exécuter seulement les modules requis pour la réalisation d'une tâche permettant ainsi un usage optimal des ressources. Les modules de ROS, appelés des *packages*, sont les outils qui nous aident à réaliser les tâches requises par notre robot. Ces *packages* peuvent exécuter des tâches de bas niveau comme communiquer avec les périphériques, combiner des sources de données et aussi peuvent réaliser des tâches plus complexes comme faire de la cartographie, planifier des trajectoires, etc. Ces modules de ROS sont développés par toute la communauté d'utilisateurs, ce qui explique la compatibilité de ROS avec un très grand nombre de matériels et périphériques. Nous travaillons avec la distribution la plus récente de ROS : ROS *Kinetic*.

La figure 1.3 présente l'architecture de notre programme implémenté dans ROS. Nous collectons les données des composants et les faisons passer dans les deux *packages* de cartographie et de planification pour qu'ils puissent générer une carte en temps réel de l'environnement et planifier la trajectoire locale et globale vers la destination. La carte et la trajectoire sont fournies ensuite aux modules de navigation qui calculent les commandes pour les moteurs. Le superviseur va prendre la décision finale pour le mode de navigation qui contrôle les mouvements de la chaise.

Pour tester rapidement les algorithmes qui seront mis en œuvre sur la chaise, nous utilisons

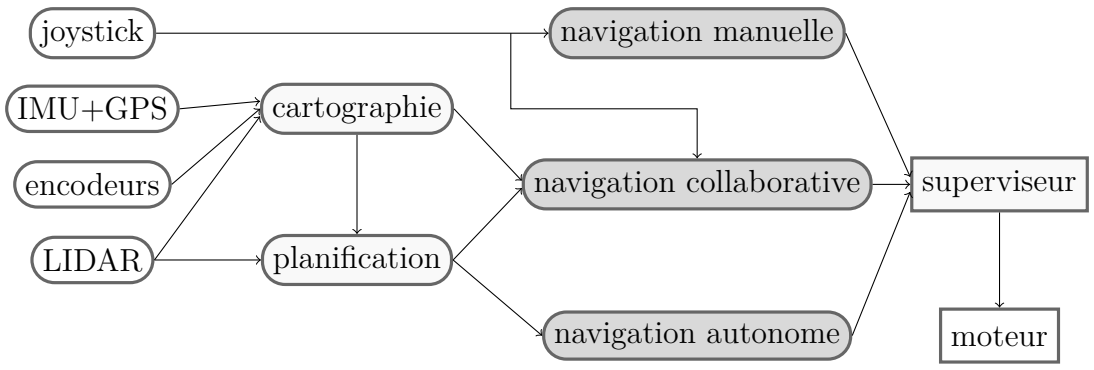


Figure 1.3 Architechture du système

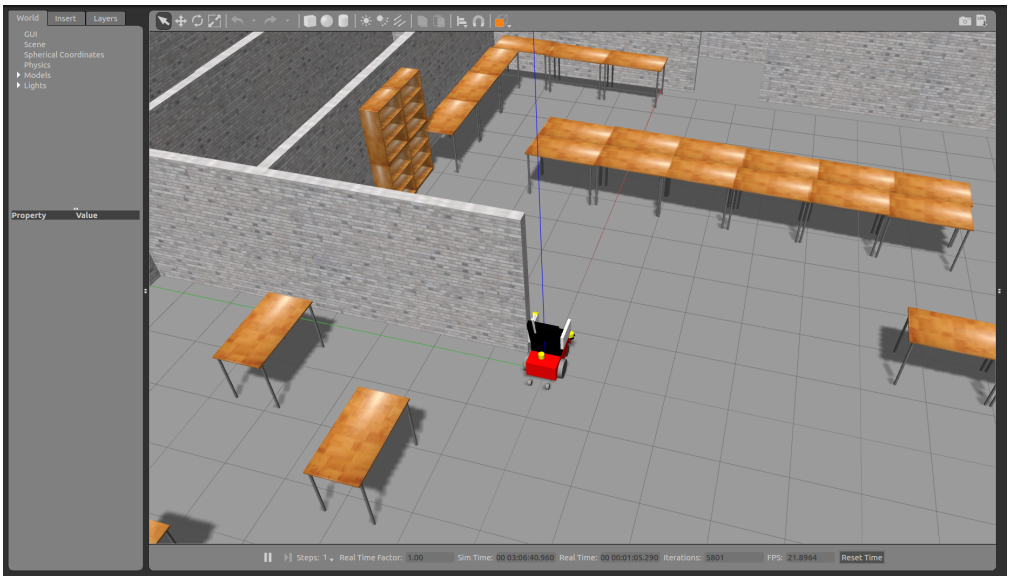


Figure 1.5 Notre fauteuil roulant dans un environnement virtuel créé et simulé avec *Gazebo*

le simulateur qui va de pair avec ROS : *Gazebo*. C'est un outil puissant qui nous permet de simuler un ou des robots dans les environnements complexes avec un moteur physique (*physics engine*), un Interface Homme-Machine (IHM) pour visualiser l'environnement virtuel et qui supporte deux types de programmation : classique et graphique. Un exemple de *Gazebo* est illustré à la figure 1.5.

RVIZ est un module de ROS qui nous permet de visualiser plusieurs types de données, par exemple : la trajectoire du robot, les repères utilisées, la carte de l'environnement, les données du capteurs (LIDAR, caméra, etc.), le modèle du robot, etc. Avec RVIZ, on peut aussi donner des commandes au robot ou à l'algorithme de contrôle, par exemple : la destination souhaitée, l'estimation de la position actuelle du robot, etc. À la figure 1.7, on peut voir le modèle du fauteuil avec la carte, la destination souhaitée, et les données du LIDAR.

1.2 Éléments de la problématique

En résumé, nos fauteuils possèdent déjà des modules de cartographie et de contrôles fonctionnels et qui sont prêts à être utiliser. Notre tâche est d'améliorer l'expérience des personnes qui l'utilisent. Nos chaises contiennent deux entités de contrôles qui peuvent s'exécuter indépendamment l'un de l'autre (les contrôles manuel et automatique). Ces deux agents ne partagent pas du tout la même source d'information. L'un utilise des données polyvalentes (visuelle, sonore, etc.) et l'autre utilise des données plus simples (LIDAR, odométrie, etc.) mais avec un plus grand niveau de certitude. La même chose se passe pour le traitement

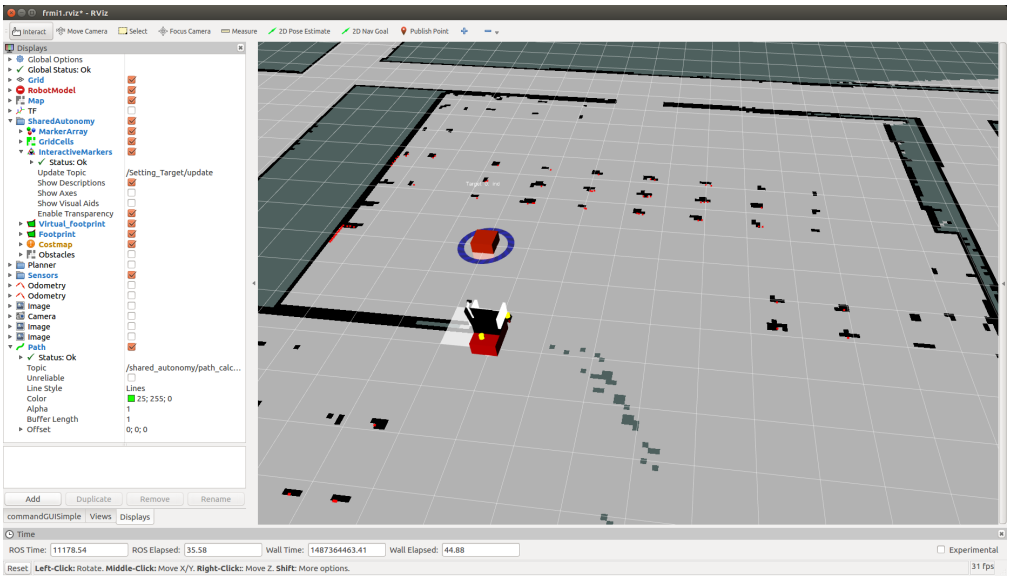


Figure 1.7 Notre fauteuil roulant dans RVIZ

des données. L'un prend une décision suivant l'habitude, l'expérience passée, les sentiments, etc. Et l'autre va faire des calculs exacts avant de donner une commande. Un autre module de commande est donc conçu pour réunir et profiter des points forts des deux agents de commandes précédents en évitant les scénarios dangereux. C'est la commande partagée ou collaborative.

Par contre, un inconvénient de la commande automatique et la commande partagée est que l'utilisateur doit déterminer en avance la ou les (dans le cas du contrôle partagé) destinations souhaitées afin de pouvoir planifier les trajectoires. C'est une tâche qui n'est pas évidente pour les utilisateurs de FRE qui sont souvent des personnes âgées qui ne sont pas habituées aux technologies.

Afin de simplifier la tâche du côté de l'utilisateur, il faut dégager les deux points importants sur lesquels la recherche est concentrée qui vise à donner une meilleure expérience utilisateur.

Détermination automatique des destinations : En reconnaissant que le contrôle partagé peut nous amener au bon endroit s'il y a des destinations pré-déterminées, comment peut-on complètement automatiser cette tâche pour aider les utilisateurs qui ne sont pas habitués aux technologies modernes ?

Amélioration de la cartographie : Pour pouvoir bien naviguer dans l'espace avec les informations les plus courantes, la meilleure façon est d'utiliser le SLAM. Par contre, pour pouvoir déterminer les destinations, il est nécessaire d'avoir une carte *a priori* de tout l'environnement. Alors, comment peut-on combiner les deux façons de cartogra-

phier, le SLAM et la localisation à partir d'une carte statique, pour nous aider dans notre application ?

C'est autour de ces deux problèmes que nous développons une solution qui facilite la navigation des utilisateurs du FRMI.

— Pour résoudre le premier problème, on commence avec la carte de l'environnement pour trouver les points « intéressants » dans la carte. L'idée est de chercher dans la carte des salles et des couloirs pour réaliser une carte topologique de l'environnement. La navigation sera ensuite réalisée à l'aide des deux cartes topologiques et la carte d'occupation du milieu.
— Pour le deuxième problème, l'idée est de charger une carte *a priori* dans l'algorithme de SLAM pour deux raisons principales : augmenter la qualité de la fermeture des boucles (*loop-closure*) [2] et avoir une référence inchangée dans plusieurs sessions de navigations différentes pour qu'on puisse placer les destinations dans cette carte. Le SLAM permet d'améliorer la navigation par l'ajout des obstacles absents dans les plans architecturaux.

1.3 Objectifs de recherche

En prenant compte de tout le matériel sur le fauteuil, des modules de ROS et des modules créés par notre équipe de recherche, notre objectif est de compléter l'automatisation du contrôle partagé, qui permet aux utilisateurs avec aucune connaissance de la technologie d'utiliser la chaise. Afin de réaliser cet objectif, il faudra passer par les étapes suivantes :

— Étudier les modules de cartographies qui existent dans ROS afin de faire une évaluation des algorithmes qui existent et de choisir un ou des algorithmes à implémenter dans notre application.
— Proposer et implémenter un module qui permet de charger une carte (soit la carte architecturale de l'environnement, soit une carte faite par le SLAM) dans l'algorithme choisi et qui remplit les conditions précédentes. Simuler et tester ce module avec le simulateur Gazebo.
— Étudier les méthodes et algorithmes qui nous permettent de déterminer les destinations potentielles dans la carte.
— Implémenter l'algorithme qui s'intègre le mieux avec le contrôle partagé. Simuler et tester ce module avec le simulateur Gazebo.

2. Fermer la boucle : Dans le cas où le robot accumule des erreurs et revient à un point déjà visité. L'algorithme doit être capable de retrouver cet endroit et boucler la trajectoire.

— Valider le fonctionnement du programme complet dans le simulateur ainsi que sur la chaise avec des sujets humains.

1.4 Plan du mémoire

Notre mémoire se divise en cinq chapitres. Nous présentons dans le deuxième chapitre une revue de littérature qui compare quelques algorithmes de SLAM existants, souvent utilisés par la communauté. En comparant ces algorithmes, nous pouvons déterminer ceux qui s'adaptent le mieux à notre application.

Dans le troisième chapitre, nous présentons la conception d'un module de chargement de carte statique dans l'algorithme de SLAM qui nous permet d'ajouter la carte *a priori* dans le programme. À la fin de ce chapitre, nous parlons des difficultés que l'on peut rencontrer durant l'utilisation d'un tel module.

Dans le quatrième chapitre, nous introduisons la méthodologie pour trouver les destinations potentielles et comparons les méthodes possibles pour la réaliser. Nous allons ensuite implémenter la solution la plus appropriée dans ROS. À la fin du chapitre, nous proposons un *design* pour que notre module puisse communiquer avec le module de contrôle partagé.

Dans chacun de ces trois chapitres, nous débutons par une courte revue de littérature pour présenter les travaux dans le domaine.

Enfin, dans la conclusion, nous résumons notre système complet avec des modules présentés en discutant des résultats en simulations et expérimentaux, des limites que nous avons constatées et des suggestions d'améliorations qui peuvent être réalisées dans l'avenir.

CHAPITRE 2 ÉVALUATION DES TECHNIQUES DU SLAM 2D

2.1 Introduction

Dans ce chapitre, nous faisons premièrement une revue littérature des techniques de SLAM existantes. Ensuite, nous allons présenter trois *packages* de SLAM 2D basés sur des capteurs LIDAR mis en œuvre sous ROS. Afin de choisir une ou des techniques qui conviennent à notre application, nous allons évaluer les deux techniques de SLAM les plus utilisées qui sont *SLAM Karto* crée par Konolige et al. (2010) et *Gmapping* crée par Grisetti et al. (2007). Une autre technique qui sera testée est *Google Cartographer* développée par Hess et al. (2016). Nous allons tester ces trois algorithmes dans un environnement virtuel dans *Gazebo* pour évaluer la performance et la précision de ces trois modules dans la section 2.3.

2.2 Revue de littérature

Le SLAM signifie le problème de localisation et cartographie simultanées. Il est également devenu le nom de la méthode utilisée par les robots mobiles qui répond au problème de SLAM, c'est-à-dire, à construire en mettant à jour une carte dans un environnement inconnu, tout en gardant une trace de leur position actuelle et leur trajectoire. C'est un problème bien étudié dans le domaine de la robotique. En effet, la raison principale qui explique l'intérêt pour ce problème est l'avantage d'avoir une carte dynamique en temps réel.

La possession d'une carte de l'environnement peut nous aider pour :

L'information : La carte fournit une source d'information importante et intuitive pour le robot et aussi pour son utilisateur sur la topologie de l'environnement, la localisation des obstacles, l'emplacement des points d'intérêt dans la trajectoire, etc.

La localisation : Pour connaître la position actuelle exacte et pour naviguer vers la destination souhaitée, l'homme ou le robot doivent connaître la carte de l'environnement autour de lui. Lors de la localisation, l'homme compare le milieu autour de lui avec la carte qu'il connaît pour trouver les points qui lui ressemblent. La même idée est appliquée pour la robotique. Le robot va comparer les informations obtenues par les capteurs (LIDAR, caméra 3D, etc.) aux caractéristiques particulières dans la carte pour ne pas être « perdu » dans l'environnement.

La navigation globale : La navigation est souvent divisée en deux étapes :

— La planification d'une trajectoire globale qui relie la position actuelle du robot et

la destination.

— La planification locale qui consiste à maintenir le robot dans la trajectoire globale en évitant les obstacles.

Comme les capteurs ne peuvent que détecter les obstacles dans une portée limitée, une carte qui contient les données générales du milieu (les obstacles fixes, les murs, etc.) est nécessaire. Afin d'obtenir une trajectoire globale qui est correcte, il est indispensable d'avoir une carte globale qui contient une bonne portion des caractéristiques majeures de l'environnement.

Avoir une carte en temps réel, qui peut nous fournir des changements de l'environnement et des obstacles dynamiques nous donne plusieurs avantages en plus :

Environnement inconnu : Souvent, les robots doivent naviguer dans des environnements inconnu d'où la carte globale de l'environnement est indisponible. Avoir un bon algorithme de SLAM va nous aider à construire une carte de l'environnement pour les prochaines utilisations et en même temps se localiser dans l'environnement.

La navigation locale : Avec seulement la carte globale et statique, il sera difficile de naviguer dans un environnement dynamique où les meubles peuvent avoir été déplacés et où les piétons bougent autour de notre robot. En connaissant ces changements, on peut planifier une trajectoire locale pour avancer au prochain *checkpoint* dans la trajectoire globale ainsi s'approcher de la destination.

En effet, le problème de SLAM est lié au problème de probabilité et d'incertitude, ce qui domine dans le domaine de la robotique. Le fait d'utiliser les caractéristiques probabilistes augmente la robustesse de l'algorithme face aux bruits de mesures et la capacité de représenter l'incertitude dans les mesures et le processus d'estimation. Presque tous les modèles utilisés sont basés sur la loi de *Bayes* (Thrun et al., 2005).

2.2.1 Historique du SLAM

Filtre de Kalman

La technique la plus populaire d'implémentation d'un filtre de *Bayes* est le filtre de Kalman (Kalman Filter (KF)) (Thrun et al., 2005). Les KF ont deux phases qui sont : la phase de prédiction et la phase de mise à jour. La phase de prédiction estime l'état du système à partir du modèle du système et des itérations précédentes. La phase de mise à jour combine l'état estimé avec les nouvelles données récoltées par les capteurs pour estimer une nouvelle fois l'état du système, cette fois plus exacte. Le résultat de cette phase, appelé postérieure, est

ensuite utilisé par la phase de prédiction pour avoir des estimations *a priori* comme illustré à la figure 2.1. Le filtre Kalman étendu (Extended Kalman Filter (EKF)) a été développé par la suite pour résoudre des problèmes non linéaires. Des tests de la convergence et de la cohérence des solutions du SLAM basées sur l'EKF sont faits par Huang and Dissanayake (2007). L'utilisation d'un filtre basé sur l'EKF peut introduire de l'instabilité dans les résultats. Si l'incertitude de l'orientation du robot est grande, l'estimateur peut prédire des résultats qui sont trop confiants et optimistes comme illustré par Castellanos et al. (2004) et Frese (2006). C'est une des raisons qui a encouragé les chercheurs à trouver d'autres techniques afin de résoudre efficacement le problème de SLAM de façon stable.

Filtre particulaire

Une autre application du filtre de *Bayes* est le filtre particulaire. Les articles de Doucet et al. (2000) et de Murphy (2000) ont introduit la méthode pour résoudre efficacement le problème en utilisant le filtre *Rao-Blackwellized*, ce qui est, jusqu'à maintenant, une des technique les plus répandues pour résoudre le problème de SLAM. Avec le filtre *Rao-Blackwellized*, la probabilité postérieure est représentée par plusieurs particules. Chaque particule possède un poids différent. Au début du programme, les particules sont générées aléatoirement autour de la position initiale du robot. Ensuite, les particules sont comparées avec l'observation donnée par les capteurs. Des valeurs de similarité pour chaque particule seront obtenues. Certaines particules sont similaires, d'autres le sont moins ou pas du tout. Pour trouver l'état actuel du système, on peut choisir la particule avec la valeur de similarité la plus grande ou bien de faire la moyenne de l'état des particules les plus similaires à l'observation. Pour échantillonner les nouvelles particules de la prochaine itération, on utilise la similarité des particules de l'itération précédente et on génère les nouvelles particules autours des particules plus fortes. La procédure du filtre particulaire est décrite à la figure 2.2. En utilisant l'hypothèse de

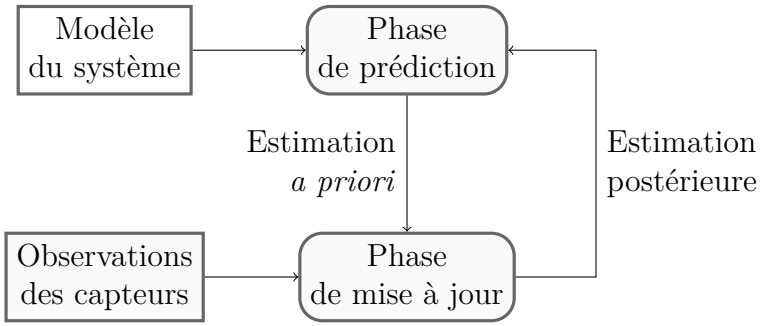


Figure 2.1 Filtre de Kalman basique

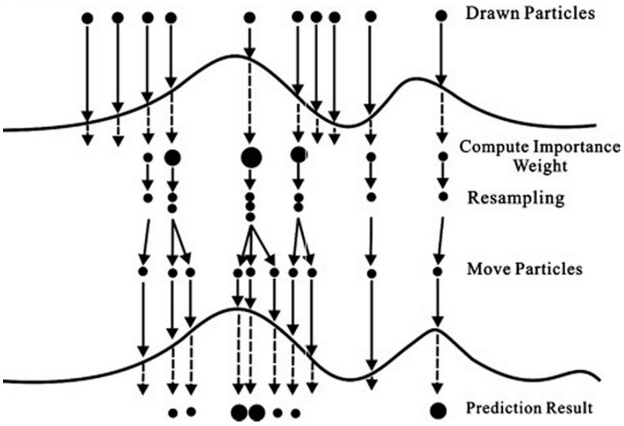


Figure 2.2 Filtre particulaire

Markov, Thrun et al. (2001), on suppose que l'état prochain du système ne dépend que de l'état précédent du système et des commandes introduites à l'entrée. L'avantage du filtre particulaire est qu'il peut traiter des incertitudes dues à une distribution multimodale et des bruits non-Gaussien du système. Les algorithmes de SLAM basés sur le filtre particulaire sont par exemples *FastSLAM* (Montemerlo et al., 2002), *Gmapping* (Grisetti et al., 2007; Grisettiyz et al., 2005), *CoreSLAM* (Steux and El Hamzaoui, 2010).

SLAM basé sur la représentation par un graphe

Une autre avenue qui est aussi importante est la solution basée sur la représentation par un graphe. Cette technique de SLAM peut surpasser des techniques de KF et du filtre particulaire dans certains aspects. Cette méthode peut manipuler des cartes avec beaucoup plus d'éléments, donc une carte plus grande (Thrun and Montemerlo, 2006). Dans cet algorithme, les données des capteurs sont utilisées pour construire un arbre de données. L'arbre est formé des nœuds et des arcs qui lient deux nœuds. Chaque nœud représente un état du robot ce qui peut contenir la pose, les données des capteurs, etc. Chaque arc représente la contrainte entre les deux nœuds qui sont deux poses successives du robot (le mouvement, une mesure, etc.). Afin d'obtenir une carte, toutes les contraintes sont linéarisées pour créer une matrice creuse [1], ce qui représente un arbre de données creux. Ce type d'algorithme a d'abord été développé par Lu and Milios (1997). Leur approche suppose que le robot est à l'arrêt pour acquérir et traiter les données avant de continuer le mouvement. Due à la procédure d'optimisation, leur algorithme n'est pas capable de résoudre le problème de SLAM à grande échelle.

1. Une matrice où presque tous les éléments sont nuls

Le problème de SLAM *offline*[2] est ensuite résolu et testé dans un environnement de grande échelle par Thrun and Montemerlo (2006) (*GraphSLAM*). Pour pouvoir réaliser la procédure d'optimisation en temps réel, une étape de réduction est ajoutée afin de simplifier l'arbre des données avant de faire l'optimisation. Cette méthode peut manipuler un nombre de caractéristiques 10^8 fois plus grand que l'ancienne méthode. *Karto SLAM*, développé par Konolige et al. (2010), peut résoudre le problème complet et en temps réel en utilisant la solution basée sur la représentation par un graphe. La technique est discutée dans la section 2.2.3. Les autres algorithmes de SLAM basés sur la représentation par un graphe sont par exemples $\sqrt{SAM}$ (Dellaert and Kaess, 2006), *iSAM* (Kaess et al., 2007), *LagoSLAM* (Carlone et al., 2012).

2.2.2 Gmapping (Grisetti et al., 2007)

Gmapping est un algorithme de SLAM utilisé avec des LIDARs. C'est un des modules de SLAM le plus utilisé dans le monde, et c'est le module de cartographie utilisé actuellement dans notre chaise. Cette méthode utilise le filtre *Rao-Blackwellized* qui est basée sur le filtre particulaire. Le problème principal du filtre *Rao-Blackwellized* est la complexité de l'algorithme. En effet, chaque particule du filtre conserve une carte de l'environnement individuelle, ce qui augmente rapidement le coût de calcul avec le nombre de particules. En revanche, plus le nombre de particules augmente, meilleure est la solution au problème. Par conséquent, le défi majeur de ce filtre est de réduire le nombre de particules en maintenant la qualité de l'estimation. De plus, l'étape d'échantillonnage des particules peut éliminer des bonnes particules du robot. Ce problème est connu comme le problème d'épuisement de particules (*particle depletion*) ou d'appauvrissement de particules (*particle impoverishmenet*) (Van Der Merwe et al., 2001).

Pour résoudre le problème et augmenter la performance du filtre *Rao-Blackwellized*, Grisetti et al. (2007) ont proposé deux méthodes pour résoudre le problème de SLAM dans l'algorithme *Gmapping* :

— Une méthode qui prend en compte la précision des capteurs et nous permet de choisir les particules plus probables, ce qui nous permet de diminuer le nombre de particules. La distribution des particules est calculée avec la pose la plus probable obtenue par la comparaison des données du LIDAR et l'information de l'odométrie. De cette façon, les informations les plus récentes sont utilisées pour générer les nouvelles particules. Cette méthode peut estimer l'état du robot avec plus de données et est donc plus

2. Le problème de SLAM hors ligne est le problème ou le robot accumule les données en faisant sa trajectoire. Et la carte est créée une fois que le robot a terminé sa mission.

précise que lorsqu'on estime seulement avec l'odométrie du robot. La carte est plus précise dans ce cas car on prend en compte la relation entre l'observation et la pose actuelle du robot avant de comparer avec les cartes individuelles.

— Une méthode est d'échantillonner adaptativement les particules afin de maintenir un nombre raisonnable de particules. Elle nous permet de faire l'échantillonnage des particules seulement si c'est nécessaire. Par conséquent, nous pouvons obtenir une carte précise en diminuant le problème d'épuisement de particules.

Le filtre *Rao-Blackwellized*

L'idée de ce filtre est d'estimer l'état du système $p(x_{1:t}, m|z_{1:t}, u_{1:t-1})$, la carte m de l'environnement et la trajectoire $x_{1:t} = x_1, ..., x_t$ du robot. L'estimation est faite en connaissant l'observation $z_{1:t} = z_1, ..., z_t$ et l'odométrie du robot $u_{1:t} = u_1, ..., u_{t-1}$. Posons la factorisation suivante :

$$p(x_{1:t}, m|z_{1:t}, u_{1:t-1}) = p(x_{1:t}|z_{1:t}, u_{1:t-1})p(m|x_{1:t}, z_{1:t}) \tag{2.1}$$

Dans l'équation (2.1), l'estimation divise le problème de SLAM en deux sous-problème distincts : l'estimation de la trajectoire $p(x_{1:t}|z_{1:t}, u_{1:t-1})$ utilisant les observations et l'odométrie du robot et la construction de la carte $p(m|x_{1:t}, z_{1:t})$ en utilisant la trajectoire estimée et les observations du robot.

le terme $p(m|x_{1:t}, z_{1:t}, u_{1:t-1})$ est simplifié par $p(m|x_{1:t}, z_{1:t})$. En effet, l'approximation peut être faite ici car la construction de la carte dépend faiblement de l'odométrie du robot. Les données d'odométrie sont utilisées pour estimer la position du robot dans le terme $p(x_{1:t}|z_{1:t}, u_{1:t-1})$. Et la carte de l'environnement est construite avec cette pose.

Cette technique de factorisation est nommée *Rao-Blackwellization*.

Avec cette approximation, le problème se divise en deux sous-problèmes :

L'estimation de la trajectoire du robot : Pour estimer la trajectoire du robot, un filtre particulaire est utilisé. Chaque particule représente une trajectoire potentielle du robot associée à une carte.

La construction de la carte : Étant donné la trajectoire et l'observation du robot, la carte postérieure peut être calculée avec la méthode de Moravec (1988). Le principe est de décomposer la surface en pixels et de l'associer à une probabilité d'occupation $p(m_i|x_{1:t}, z_{1:t})$. En faisant l'hypothèse que toutes les probabilités des pixels sont indépendantes les unes des autres, et en utilisant le théorème de *Bayes*, on peut écrire :

$$\frac{p(m_i|x_{1:t}, z_{1:t})}{1 - p(m_i|x_{1:t}, z_{1:t})} = \frac{p(m_i|x_{1:t-1}, z_{1:t-1})}{1 - p(m_i|x_{1:t-1}, z_{1:t-1})} \frac{p(m_i|x_t, z_t)}{1 - p(m_i|x_t, z_t)} \frac{1 - p(m_i)}{p(m_i)} \tag{2.2}$$

où $p(m_i)$ est la probabilité préalable de chaque pixel. Posons $L_{t,i} = \log \frac{p(m_i|x_{1:t}, z_{1:t})}{1-p(m_i|x_{1:t}, z_{1:t})}$. On a alors la formule d'itération suivante :

$$L_{t,i} = L_{t-1,i} + \log \frac{p(m_i|x_t, z_t)}{1 - p(m_i|x_t, z_t)} - \log \frac{p(m_i)}{1 - p(m_i)} \tag{2.3}$$

La probabilité $p(m_i|x_t, z_t)$ dépend du capteur utilisé dans la cartographie. La valeur de cette probabilité est la précision de l'observation. Plus l'observation est exacte, plus la vitesse de convergence de la probabilité est grande.

Le filtre particulaire le plus commun est le filtre Sampling Importance Resampling (SIR). En général, pour traiter les données des capteurs dès qu'elles sont disponibles et mettre à jour la trajectoire et la carte du robot, l'algorithme du filtre *Rao-Blackwellized* SIR suit les quatre étapes suivantes :

Échantillonnage (*Sampling*) : Les nouvelles particules $x_t^{(i)}$ sont créées à partir des anciennes particules $x_{t-1}^{(i)}$ en échantillonnant avec une distribution préliminaire[3] π. Normalement, un modèle probabiliste du mouvement du robot est utilisé pour l'échantillonnage. Par contre, cette approche nous demande de réévaluer cette distribution π à chaque fois que l'on obtient une nouvelle observation, ce qui peut être coûteux. De plus, comme la taille de la trajectoire grossit au cours du temps, la méthode devient inefficace. D'après Andrieu et al. (2001), la distribution π peut être écrite sous la forme récursive suivant :

$$\pi(x_{1:t}|z_{1:t}, u_{1:t-1}) = \pi(x_1|x_{1:t}, z_{1:t}, u_{1:t-1})\pi(x_{1:t-1}|z_{1:t-1}, u_{1:t-2}) \tag{2.4}$$

Distribution des poids aux particules (*Importance weighting*) : À chaque particule est distribué un poids $w_t^{(i)}$ qui est initialisé suivant l'importance de l'état successeur et de la distribution π. Il prend en compte le fait que la distribution préliminaire π et l'importance de l'état précédent ne sont pas égaux.

$$w_t^{(i)} = \frac{p(x_{1:t}^{(i)}|z_{1:t}, u_{1:t-1})}{\pi(x_{1:t}^{(i)}|z_{1:t}, u_{1:t})} \tag{2.5}$$

Basé sur les deux équations (2.4) et (2.5), on peut calculer le poids $w_t^{(i)}$ avec la formule

3. *Proposal distribution* en anglais

de récurrence suivante :

$$w_t^{(i)} = \frac{p(x_{1:t}^{(i)}|z_{1:t}, u_{1:t-1})}{\pi(x_{1:t}^{(i)}|z_{1:t}, u_{1:t})} = \frac{\eta p(z_t|x_{1:t}^{(i)}, z_{1:t-1}) p(x_t^{(i)}|x_{t-1}^{(i)}, u_{t-1})}{\pi(x_1^{(i)}|x_{1:t}^{(i)}, z_{1:t}, u_{1:t-1})} \underbrace{\frac{p(x_{1:t-1}^{(i)}|z_{1:t-1}, u_{1:t-2})}{\pi(x_{1:t-1}^{(i)}|z_{1:t-1}, u_{1:t-2})}}_{w_{t-1}^{(i)}} \tag{2.6}$$

Le terme $\eta = 1/p(z_t|z_{1:t-1}, u_{1:t-1})$ dans l'équation (2.6) est un facteur de normalisation et qui est constant pour toutes les particules.

Ré-échantillonnage (*Resampling*) : On redistribue les particules suivant les poids calculés. Après le ré-échantillonnage, toutes les particules ont le même poids. Cette étape est nécessaire pour limiter le nombre des particules dans l'algorithme et pour repositionner les particules dans le cas où la distribution préliminaire π et l'importance de l'état précédent ne sont pas égaux.

Estimation de la carte (*Map estimation*) : Pour chaque particule, une carte $p(m^{(i)}|x_{1:t}^{(i)}, z_{1:t})$ est estimée en se basant sur la trajectoire de cette particule $x_{1:t}^{(i)}$ et les observations reçues $z_{1:t}$. C'est un problème de cartographie avec trajectoire connue.

Distribution préliminaire améliorée

En utilisant le modèle probabiliste du mouvement du robot pour établir la distribution préliminaire π, on a utilisé une solution sous-optimale qui nous force à échantillonner un nombre de particules plus grand que néccessaire. En utilisant les LIDAR, la précision des capteurs est meilleure que celle de l'odométrie (ce qui est souvent le cas dans le SLAM). En prenant compte de l'observation dans la distribution préliminaire, on peut concentrer l'échantillonnage dans la zone de vraisemblance de l'observation. Doucet (1998) a proposé une formule de distribution qui est optimale par rapport aux variances des poids des particules :

$$p(x_t|m_{t-1}^{(i)}, x_{t-1}^{(i)}, z_t, u_{t-1}) = \frac{p(z_t|m_{t-1}^{(i)}, x_t) p(x_t|x_{t-1}^{(i)}, u_{t-1})}{p(z_t|m_{t-1}^{(i)}, x_{t-1}^{(i)}, u_{t-1})} \tag{2.7}$$

En réalité, si la vraisemblance de l'observation est grande, le nombre de particules est grand. On observe donc le problème suivant : il est nécessaire d'évaluer beaucoup de particules pour couvrir une petite région. En outre, si le capteur est beaucoup plus précis que l'odométrie, le terme $p(z_t|m_{t-1}^{(i)}, x_t)$ domine dans la formule de la distribution optimale, et dans la plupart des cas, on a un seul maximum comme illustré à la figure 2.3. Alors, l'échantillonnage peut être réalisé seulement autour de ce maximum. En ignorant les régions moins probables, on échantillonne moins de particules, ce qui consomme moins de ressources. *Gmapping* propose

une technique pour bien échantillonner les particules :

— Évaluer la probabilité de l'observation avant de faire l'échantillonnage. La zone de probabilité maximum sera utilisée pour l'échantillonnage.
— Échantillonner K particules autour de ce maxima $L^{(i)} = \{x|p(z_t|m_{t-1}^{(i)}, x) > \epsilon\}$. Chacune des particules possède les paramètres $\mu_t^{(i)}$ et $\Sigma_t^{(i)}$ individuels dans lesquels l'odométrie du robot est prise en compte :

$$\mu_t^{(i)} = \frac{1}{\eta^{(i)}} \sum_{j=1}^{K} x_j p(z_t|m_{t-1}^{(i)}, x_j) p(x_j|x_{t-1}^{(i)}, u_{t-1}) \tag{2.8}$$

$$\Sigma_t^{(i)} = \frac{1}{\eta^{(i)}} \sum_{j=1}^{K} p(z_t|m_{t-1}^{(i)}, x_j) p(x_j|x_{t-1}^{(i)}, u_{t-1})(x_j - \mu_t^{(i)})(x_j - \mu_t^{(i)})^T \tag{2.9}$$

avec le facteur de normalisation η :

$$\eta^{(i)} = \sum_{j=1}^{K} p(z_t|m_{t-1}^{(i)}, x_j) p(x_j|x_{t-1}^{(i)}, u_{t-1}) \tag{2.10}$$

En utilisant cette méthode, le poids des particules devient :

$$w_t^{(i)} \approx w_{t-1}^{(i)} \eta^{(i)} \tag{2.11}$$

En utilisant cette technique, la distribution Gaussienne de chaque particule est déterminée séparément. De plus, la distribution prend en compte simultanément l'odométrie et l'observation du robot en limitant le nombre de particules à échantillonner. Les résultats sont plus probables que dans le cas où on considère seulement l'odométrie du robot. Le poids des particules est ensuite calculé efficacement avec une formule récursive.

Ré-échantillonnage adaptatif

Une autre problème commun des filtres particulaires est l'appauvrissement des particules dans l'étape de ré-échantillonnage. Dans cette étape, les particules de poids faibles sont remplacées par celles de poids élevés. Puisqu'on utilise un nombre fini de particules, cette étape est nécessaire dans le cas où la distribution préliminaire et la distribution postérieure des particules diffèrent. En revanche, le ré-échantillonnage peut enlever de bonnes particules et peut faire diverger l'algorithme. Pour résoudre ce problème, il faut trouver un bon critère pour décider quand on fait le ré-échantillonnage. Liu and Chen (1998) ont proposé le concept du nombre de particules effectives pour estimer la qualité de l'ensemble des échantillons. *Gmapping* utilise la formule de Andrieu et al. (2001) pour calculer cette valeur :

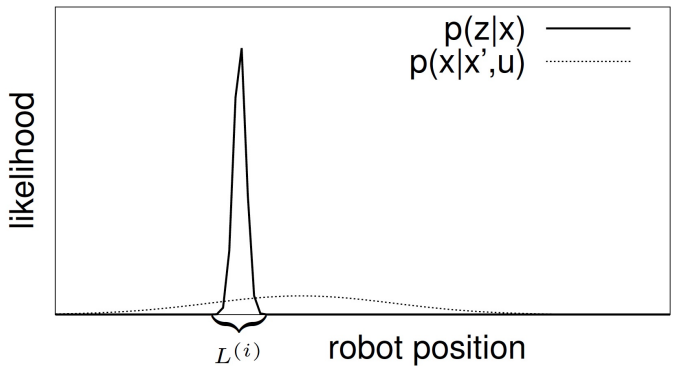


Figure 2.3 Deux composants du modèle de mouvement. Dans l'intervalle $L^{(i)}$, la probabilité de l'observation $p(z_t|m_{t-1}^{(i)}, x_t)$ domine le produit $p(z_t|m_{t-1}^{(i)}, x_t)p(x_t|x_{t-1}^{(i)}, u_{t-1})$. (Grisetti et al., 2007)

$$N_{\mathit{eff}} = \frac{1}{\sum_{i=1}^{N}(w^{(i)})^2} \tag{2.12}$$

L'idée est d'évaluer la variance des poids des particules. La bonne approximation de l'ensemble des particules donne une petite variance. Suivant l'algorithme proposé par Andrieu et al. (2001), le ré-échantillonnage est fait chaque fois que N_{eff} est plus petit que $N/2$ avec N le nombre de particules souhaité. Cette façon de ré-échantillonner réduit le risque d'appauvrissement de particules car le ré-échantillonnage est fait seulement si nécessaire.

Algorithme

Le processus de l'algorithme *Gmapping* est précisé à l'annexe A. En résumé, chaque fois qu'une mesure (u_{t-1}, z_t) est obtenue, la distribution préliminaire est calculée et les particules sont mises à jour. Les étapes sont les suivantes :

1. Déduire les poses préliminaires du robot $x_t^{'(i)}$ avec les données d'odométrie u_{t-1} et la pose précédente $x_{t-1}^{(i)}$.
2. Comparer les cartes obtenues par ces poses $m_{t-1}^{(i)}$ avec les observations des capteurs z_t. La comparaison est limitée autour de la position préliminaire $x_t^{'(i)}$. Si la comparaison échoue, la pose et le poids de l'échantillon sont calculés avec le modèle de mouvement (les étapes 3 et 4 sont ignorées). Si la comparaison est réussie, on prend la pose la plus probable dans la zone autour de la position préliminaire $\hat{x}_t^{(i)}$.
3. Un ensemble de l'échantillon est choisi autour de $\hat{x}_t^{(i)}$. La moyenne et la covariance de ces points x_j sont calculées avec les équations (2.8) et (2.9). Le facteur de normalisation

$\eta^{(i)}$ est aussi calculé avec (2.10).

4. Les nouvelles poses $x_t^{(i)}$ des particules i sont prises à partir de la distribution Gaussienne $N(\mu_t^{(i)}, \Sigma_t^{(i)})$ de la distribution préliminaire améliorée.
5. Mise à jour du poids des nouvelles particules.
6. Les cartes $m^{(i)}$ sont mises à jour avec les nouvelles poses $x_t^{(i)}$ et l'observation obtenue z_t.
7. Si $N_{\mathit{eff}} < N/2$, ré-échantillonnage.

2.2.3 KartoSLAM (Konolige et al., 2010)

KartoSLAM est un algorithme basé sur la représentation par un graphe qui a été développé par *SRI International's Karto Robotics* et ensuite implémenté dans ROS en utilisant la méthode de décomposition de matrice de Cholesky pour résoudre les systèmes creux et linéaires. C'est une méthode qui est bien optimisée et non-itérative (seulement une itération de l'algorithme est faite à chaque observation du robot). Le SLAM basé sur la représentation par un graphe utilise un arbre pour représenter le système. *KartoSLAM* utilise l'arbre de poses pour représenter leur problème de SLAM. Dans l'arbre de poses, les nœuds représentent les poses du robots c_i le long de la trajectoire et ses observations des capteurs S_i. Ils sont connectés par des contraintes non-linéaires z_{ij} obtenues en connectant les mouvements entre les nœuds successifs ou en comparant les observations des nœuds proches avec la fermeture des boucles. La figure 2.4 nous montre un exemple du graphe avant et après les fermetures des boucles qui ajoutent les deux contraintes qui lient les nœuds (c_1, S_1), (c_3, S_3) et (c_0, S_0), (c_4, S_4). Toutes les contraintes dans le graphe seront ensuite prises en compte pour l'optimisation afin d'obtenir une trajectoire du robot et la carte de l'environnement. L'optimisation des grands arbres de poses est toujours un problème dans le domaine car le temps de calcul grandit de

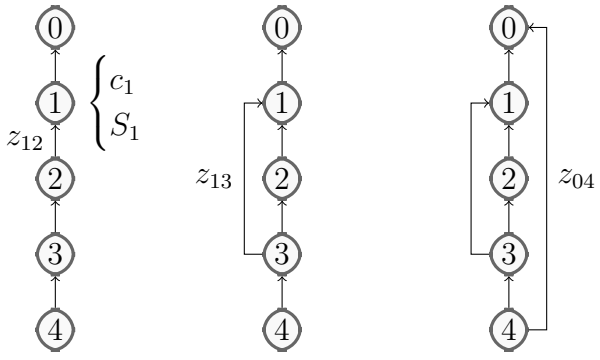


Figure 2.4 Le SLAM basé sur la représentation par un graphe

façon cubique en fonction de la taille de l'arbre. *KartoSLAM* résout efficacement ce problème en le transformant en un sous-problème linéaire (Sparse Pose Adjustment (SPA)). Le SPA est extrêmement efficace dans le cas de cartographie dans un grand environnement car il maintient seulement l'arbre de poses du robot.

Dans les algorithmes de SLAM basé sur la représentation par un graphe, il existe deux problèmes principaux :

— Le premier problème est de bien identifier les contraintes avec les observations. La difficulté de ce problème est l'ambiguïté ou la symétrie dans l'environnement. Une solution pour résoudre ce problème est de traiter directement les données des capteurs (*front-end*).

— Le deuxième problème est de bien corriger les poses du robot afin d'obtenir une carte précise et stable de l'environnement en connaissant toutes les contraintes. Cette partie consiste à bien optimiser le SLAM (*back-end*). L'algorithme doit chercher une configuration des nœuds qui maximise la cohérence des mesures avec les contraintes.

Dans les algorithmes basés sur la représentation par un graphe, l'exécution est la combinaison de *front-end* et *back-end* qui est illustrée à la figure 2.5. Ceci est nécessaire car la procédure *front-end* doit être exécutée en temps réel. La carte partielle où on améliore avec chaque observation doit donc être optimisé par le traitement *back-end*. L'algorithme SPA est utilisée dans les deux phases *front-end* et *back-end* du programme *KartoSLAM*.

Méthode de LM(Levenberg, 1944; Marquardt, 1963)

Pour optimiser l'ensemble des poses et des contraintes, *KartoSLAM* utilise la méthode de Levenberg-Marquardt. LM est un algorithme pour résoudre les problèmes des moindres carrés non-linéaires. C'est une méthode basée sur l'algorithme de Gauss-Newton (GN) et l'algorithme du gradient. Cet algorithme permet de trouver seulement le minimum local du

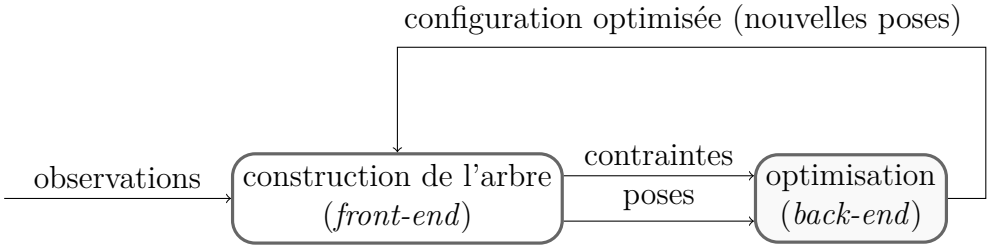


Figure 2.5 Système des SLAM basés sur la représentation par un graphe

problème. La solution initiale doit donc être proche de la solution globale du problème.

Le problème est le suivant :

Soit l'ensemble de m paires de données (x_i, y_i). On cherche le paramètre $\boldsymbol{\beta}$ du modèle $f(x, \boldsymbol{\beta})$ qui minimise la fonction :

$$\hat{\boldsymbol{\beta}} = \arg\min_{\beta} \sum_{i=1}^{m} [y_i - f(x_i, \boldsymbol{\beta})]^2 \tag{2.13}$$

Comme les autres procédures d'optimisation numériques, LM est un algorithme itératif. On choisit d'abord la solution initiale $\boldsymbol{\beta}_0$. Pour chaque itération, le paramètre $\boldsymbol{\beta}$ est remplacé par $\boldsymbol{\beta}_{j+1} = \boldsymbol{\beta}_j + \boldsymbol{\delta}$. La fonction $f(x_i, \boldsymbol{\beta} + \boldsymbol{\delta})$ est approximée par :

$$f(x_i, \boldsymbol{\beta} + \boldsymbol{\delta}) \approx f(x_i, \boldsymbol{\beta}) + \boldsymbol{J_i}\boldsymbol{\delta} \tag{2.14}$$

avec $\boldsymbol{J_i} = \frac{\partial f(x_i, \beta)}{\partial \boldsymbol{\beta}}$ le gradient de la fonction f par rapport au $\boldsymbol{\beta}$. La solution de LM est :

$$(\boldsymbol{J^T J} + \lambda \mathrm{diag}(\boldsymbol{J^T J}))\boldsymbol{\delta} = \boldsymbol{J^T}[\boldsymbol{y} - f(\boldsymbol{\beta})] \tag{2.15}$$

où λ est un facteur d'amortissement qui est ajusté à chaque itération. Si la vitesse de convergence de l'algorithme est grande, λ est choisi petit pour ramener la formule à celle de GN : $\boldsymbol{J^T J\delta} = \boldsymbol{J^T}[\boldsymbol{y} - f(\boldsymbol{\beta})]$. Et inversement, si la vitesse de convergence de l'algorithme est lente, λ est choisi grand pour que l'itération prenne la direction du gradient de la fonction f. L'algorithme s'arrête si le pas δ devient négligeable ou si $\hat{\boldsymbol{\beta}}$ a atteint la précision souhaitée. L'algorithme LM peut résoudre le problème de la méthode GN où la fonction possède plusieurs points d'inflexions avec une vitesse de convergence similaire et avec une estimation initiale moins précise.

La formulation du problème LM

En se basant sur l'algorithme *Sparse Bundle Adjustment* de la vision par ordinateur, qui est aussi une implémentation efficace de LM, (Konolige et al., 2010) propose leur propre système pour la cartographie 2D qui est le *Sparse Pose Adjustment*.

La variable du système est la pose $c_i = [t_i, \theta_i]^T = [x_i, y_i, \theta_i]^T$ du robot, correspondant à une translation et un angle [4]. La contrainte entre les deux nœuds de l'arbre est la mesure d'un nœud par rapport à l'autre. La différence entre les deux nœuds c_i et c_j dans le repère i

4. Notons que tous les angles dans *KartoSLAM* sont normalisé dans l'intervalle $(-\pi, \pi]$

est z_{ij} avec la précision Λ_{ij} [5] et la matrice de rotation de c_i est R_i. L'équation de mesure qui décrit la différence entre les deux poses peut être exprimée comme :

$$h(c_i, c_j) = \begin{cases} R_i^T(t_j - t_i) \\ \theta_j - \theta_i \end{cases} \tag{2.16}$$

La fonction qui calcule l'erreur d'une contrainte e_{ij} et l'erreur total du système $\chi^2(\boldsymbol{c}, \boldsymbol{e})$ est :

$$e_{ij} = z_{ij} - h(c_i, c_j) \tag{2.17}$$

$$\chi^2(\boldsymbol{c}, \boldsymbol{e}) = \sum_{ij} e_{ij}^T \Lambda_{ij} e_{ij} \tag{2.18}$$

La configuration optimale pour l'ensemble des poses $\boldsymbol{c}$ est celle qui minimise l'erreur totale dans l'équation (2.18). On peut maintenant formuler la solution du problème d'optimisation LM en utilisant l'équivalence suivante entre les variables :

Tableau 2.1 Équivalence entre les variables de *KartoSLAM* et le problème LM

	Problème LM	*KartoSLAM*
Solution cherchée	$\boldsymbol{\beta}$	$\boldsymbol{c}$
Première estimation	$\boldsymbol{\beta}_0$	$\boldsymbol{c}$ actuel
Pas d'itération	δ	$\boldsymbol{c} - c_0$
Fonction à minimiser	$f(\boldsymbol{x}, \beta)$	$\chi^2(\boldsymbol{c}, \boldsymbol{e})$

Posons $\boldsymbol{J} = \frac{\partial \boldsymbol{e}}{\partial \boldsymbol{x}}$ et $\boldsymbol{H} = \boldsymbol{J}^{\boldsymbol{T}} \boldsymbol{\Lambda} \boldsymbol{J}$. Le système LM est alors :

$$(\boldsymbol{H} + \lambda \text{diag} \boldsymbol{H}) \Delta x = \boldsymbol{J}^{\boldsymbol{T}} \boldsymbol{\Lambda} \boldsymbol{e} \tag{2.19}$$

Structure de la matrice creuse $\boldsymbol{H}$

Le nombre des variables d'état est trois fois la cardinalité de $\boldsymbol{c}$. Pour un grand nombre de poses, avec cette approche, le nombre de variables dans ce système est conséquent. Par contre, dans notre cas, le nombre de contraintes augmente seulement linéairement avec le nombre de poses car les contraintes existent habituellement seulement entre deux poses successives.

5. L'algorithme SPA demande la matrice de précision $\boldsymbol{\Lambda}$ estimée avec l'algorithme de comparaison des cartes (*scan-matching*), par exemple, Konolige and Chou (1999), Olson (2009)

La matrice Λ est souvent donc une matrice diagonale par blocs 3×3, ce qui résulte en une matrice $\boldsymbol{H}$ creuse. On peut alors résoudre le problème plus rapidement.

Pour résoudre (2.19) en format creux, *KartoSLAM* utilise le module *CSparse* (Davis, 2006) qui est un module optimisé utilisant la décomposition de Cholesky pour résoudre les systèmes creux. *CSparse* utilise le format de Compressed Column Storage (CCS) pour stocker la matrice creuse. L'idée est d'utiliser trois vecteurs qui vont sauvegarder les indices des colonnes (*col_ ptr*), des rangées (*col_ ptr*) et la valeur (*val*) des éléments non nuls dans la matrice. Les vecteurs sont ordonnés selon les colonnes puis les rangées. La dernière valeur du vecteur *col_ ptr* est le nombre total des éléments non nuls dans la matrice creuse.

Le format CCS est efficace en stockage. En revanche, il est difficile de créer la matrice sous cette forme car chaque élément ajouté nous demande de décaler les trois vecteurs. La meilleure façon de la créer est d'établir la matrice creuse $\boldsymbol{H}$ selon l'ordre des colonnes. Ensuite, les valeurs sont mises dans les vecteurs du format CCS. L'algorithme de cette technique est précisé à l'Annexe B. Notons que seuls les éléments dans la partie supérieure de $\boldsymbol{H}$ sont conservés, car la méthode de Cholesky considère que la matrice est symétrique et traite le triangle supérieur de la matrice.

Le système de LM avec λ persistant

L'algorithme de LM avec λ persistant est présenté dans l'Annexe C. Dans chaque appel de fonction, il calcule une itération de l'algorithme LM avec l'ensemble des nœuds associés à l'observation. Le fait d'exécuter une seule itération à chaque fois nous permet d'ajouter les nœuds à l'algorithme au fur et à mesure. L'algorithme assure la continuité de λ tout au long des exécutions de LM, et les nœuds ajoutées peuvent changer λ dès qu'ils sont rajoutés. L'algorithme est basé sur l'idée que le changement fait par quelques nœuds n'est pas significatif. L'utilisation d'un λ qui varie en fonction de l'itération précédente permet de continuer l'optimisation dans un mode similaire (GN ou gradient). Ceci peut aider dans le cas de la fermeture des boucles.

2.2.4 Google Cartographer (Hess et al., 2016)

Cartographer est un algorithme de SLAM en 2D et en 3D basé sur la représentation par un graphe et développé par *Google*. Il est implémenté en C++ afin de supporter plusieurs plateformes et plusieurs configurations de capteurs. Il existe aussi un module d'intégration en ROS pour *Cartographer*. La contribution de *Google* dans l'algorithme de SLAM est une méthode efficace et précise pour faire la fermeture des boucles. Avec cette façon de diminuer

la complexité du programme, la technique nous permet de faire le SLAM en temps réel dans un grand environnement.

Afin de pouvoir réaliser la fermeture des boucles plus efficacement, *Google* a proposé un changement dans la structure du SLAM basé sur la représentation par un graphe classique afin de rendre leur méthode d'optimisation possible. Ils ont introduit les sous-cartes (*submaps*) qui sont une grille de probabilité créée à partir de quelques nœuds successifs.

La façon classique de faire un SLAM est de comparer nœud par nœud dans les cartographies locale et globale pour calculer les changements de poses du robot (Olson, 2015; Konolige et al., 2010; Lu and Milios, 1997). Par contre, cette façon peut facilement accumuler des erreurs. Dans le cas de *Cartographer*, deux stratégies différentes sont utilisées pour faire les cartographies locale et globale.

— Pour la cartographie locale, les données des capteurs sont insérées dans les sous-cartes en estimant les meilleures poses, ce qui est suffisant pour une petite période de temps. Les comparaisons sont faites dans la sous-carte récente seulement, ce qui peut optimiser la puissance de calcul car il ne dépend que des données récentes. Les erreurs accumulées dans cette phase de cartographie seront éliminées dans la deuxième phase d'optimisation globale du système.
— Pour la cartographie globale, une optimisation des poses est faite toutes les quelques secondes. Une fois que la sous-carte est complète (on n'ajoute plus de données dans cette sous-carte), elle est prise en compte pour la fermeture de boucle. Toutes les sous-cartes qui sont terminées sont aussi prises en compte. Si elles sont assez proches, basées sur les estimations courantes, une comparaison est faite pour essayer de mettre les poses et les sous-cartes ensemble. Si un bon couple est trouvé dans la zone autour de la pose, une contrainte d'optimisation est ajoutée. En faisant cette opération régulièrement, on peut reconnaître un lieu dès qu'on fait le retour à cet endroit. Cette opération doit être plus rapide que la cartographie locale. Sinon, la fermeture de boucle sera retardée.

Définition et éléments de base

Dans cette section, on définit les symboles mathématiques utilisés dans la formulation des problèmes des sections suivantes :

— M est la grille de probabilité des sous-cartes où la probabilité des pixels est limitée dans l'intervalle $[p_{min}, p_{max}]$.
— K est le nombre d'acquisitions du LIDAR.

— d_{max} est la portée utilisable maximale du LIDAR
— $H = \{h_k\}_{k=1,...K}, h_k \in R^2$ est la donnée du LIDAR de chaque acquisition.
— ξ est une pose : position $[\xi_x, \xi_y]^T$ et angle ξ_θ.
— T_ξ est la transformation de la pose ξ dans le repère de la sous-carte, avec la translation t_ξ et la rotation R_ξ. La position des observations du LIDAR peut être calculée avec $T_\xi h_k$.
— Σ_{ij} est la matrice de covariance de la pose ξ_{ij} du robot dans la sous-carte M_i.
— p_{hit} et p_{miss} sont les probabilités d'un point dans la grille de probabilité qui est occupé (*hits*) ou libre (*misses*). À chaque récupération de données, on insère des points observés dans l'ensemble des points occupés. Tous les points au milieu de la trajectoire (de l'origine des rayons au point observé) sont dans l'ensemble des points libres. La procédure est décrite à la figure 2.6. La mise à jour de la grille de probabilité M peut se faire avec l'itération :

$$odds(p) = \frac{p}{1-p} \tag{2.20}$$

$$M_{new} = clamp\left(odds^{-1}\left(odds\left(M_{old}\left(x\right)\right) odds\left(p_{hit}\right)\right), p_{min}, p_{max}\right) \tag{2.21}$$

La cartographie locale

La cartographie locale est faite à l'aide de la librairie *Ceres* (Agarwal et al.) qui est une librairie pour résoudre des problèmes des moindres carrés non-linéaires développée aussi par *Google*. Dans ce module de SLAM 2D, nous utilisons *Ceres* avec la méthode LM pour résoudre le problème, ce qui donne une solution locale. Une bonne estimation initiale de position est tout de même nécessaire. L'algorithme est responsable d'optimiser la probabilité de la pose du robot dans la carte. La fonction à optimiser est :

$$\arg\min_{\xi} \sum_{k=1} K\left(1 - M_{smooth}\left(T_\xi h_k\right)\right)^2 \tag{2.22}$$

Ici, la fonction M_{smooth} est une version lisse de la probabilité du point dans la sous-carte. En utilisant l'interpolation, la valeur de probabilité peut sortir de l'intervalle $[0, 1]$ mais il n'affecte pas le résultat final. En lissant la fonction de probabilité, l'opération d'optimisation donne une meilleure précision que la résolution de la carte.

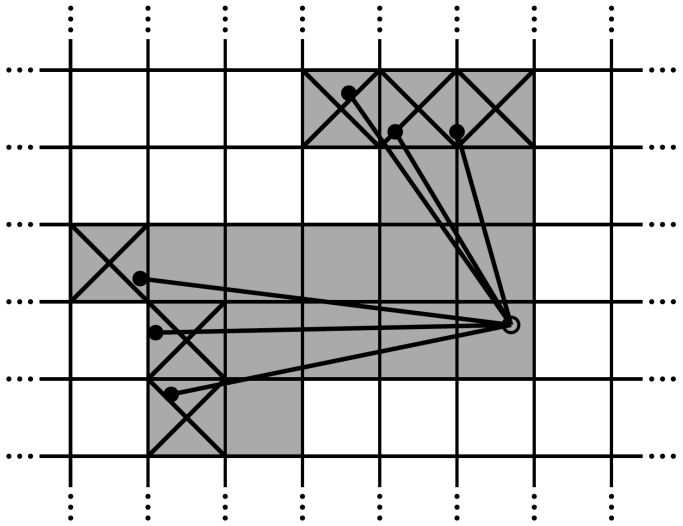


Figure 2.6 Observation récupérée et associée dans les grilles d'occupation (*hits* en gris et barré, *misses* en gray, les pixels blancs ne sont pas observés (*unknown*)) (Hess et al., 2016)

La cartographie globale

Pour traiter un large environnement, puisque la phase cartographie locale accumule des erreurs qui sont importantes après un certain nombre d'observations, une optimisation globale est nécessaire pour limiter ces erreurs. L'approche de *Cartographer* est basée sur celle de *KartoSLAM*. Toutes les poses sont sauvegardées et utilisées pour l'optimisation globale. En outre, les sous-cartes complétées sont aussi sauvegardées pour faire la fermeture de boucle. Cette procédure fonctionne en arrière-plan, et dès qu'une bonne correction est trouvée, elle est ajoutée dans le problème d'optimisation globale.

Le problème d'optimisation globale est aussi un problème des moindres carrés non linéaire. Avec les données calculées dans la phase locale de l'algorithme à partir de l'observation, la fonction à optimiser avec *Ceres* (Agarwal et al.) est :

$$e\left(\xi_i^m, \xi_j^s, \xi_{ij}\right) = \xi_{ij} - \begin{pmatrix} R_{\xi_i^m}^{-1}(t_{\xi_i^m} - t_{\xi_j^s}) \\ \xi_{i,\theta}^m - \xi_{j,\theta}^s \end{pmatrix} \tag{2.23}$$

$$E^2\left(\xi_i^m, \xi_j^s, \Sigma_{ij}, \xi_{ij}\right) = e\left(\xi_i^m, \xi_j^s, \xi_{ij}\right)^T \Sigma_{ij}^{-1} e\left(\xi_i^m, \xi_j^s, \xi_{ij}\right) \tag{2.24}$$

$$\arg\min_{\Xi^m, \Xi^s} \frac{1}{2} \sum_{ij} \rho\left(E^2\left(\xi_i^m, \xi_j^s, \Sigma_{ij}, \xi_{ij}\right)\right) \tag{2.25}$$

où $\Xi^m = \xi_i^m$ et $\Xi^s = \xi_j^s$ sont les ensembles des poses des sous-cartes (***m**aps*) et des

observations (***s**cans*) dans le repère global. L'optimisation de la fonction (2.25) minimise l'erreur globale du système avec une fonction qui réduit l'influence des points incorrects, par exemple quand la procédure locale ajoute une contrainte erronée dans le système, ce qui peut exister dans les environnements symétriques (Agarwal et al., 2013).

Branch and bound

Pour pouvoir trouver les contraintes à ajouter dans le problème d'optimisation, on doit être capable de trouver le bon couple de pose et sous-carte à insérer dans la fonction (2.25). Une recherche autour de la pose est faite pour trouver d'autres sous-cartes qui peuvent contenir la pose. Pour y arriver, une comparaison pixel par pixel est faite afin de trouver la probabilité maximale de l'observation (avec la pose optimale ξ^*) dans les sous-cartes autour de la région de recherche W.

$$\xi^* = \arg\max_{\xi \in W} \sum_{k=1}^{K} M_{nearest}(T_\xi h_k) \tag{2.26}$$

Ici, la fonction $M_{nearest}$ fait l'arrondi des observations au pixel le plus proche avant de calculer la probabilité. La recherche est faite dans la région W. Pour bien choisir cette région, il faut assurer que le pas angulaire δ_θ ne fait pas dévier la sous-carte plus que r qui est la résolution de la carte (la taille d'un pixel). On a alors :

$$d_{max} = \max_{k=1,\ldots,K} \|h_k\| \tag{2.27}$$

$$\delta_\theta = \arccos\left(1 - \frac{r^2}{2d_{max}^2}\right) \tag{2.28}$$

La région W est alors tous les pas dans la zone souhaitée ($W_x = W_y = 5\,\text{m}$ et $W_\theta = 30°$ par exemple). $\bar{W}$ est un sous-ensemble discret de $W_x \in [-w_x, w_x]$, $W_y \in [-w_y, w_y]$ et $W_\theta \in [-w_\theta, w_\theta]$

$$W = \xi_0 + (rj_x, rj_y, \delta_\theta j_\theta) : (j_x, j_y, j_\theta) \in \bar{W} \tag{2.29}$$

Pour trouver la pose optimale ξ^*, un balayage de toutes les possibilités dans W prend trop de temps. Afin de trouver ξ^* d'une façon efficace et rapide, l'algorithme de séparation et évaluation (*Branch and bound*) est utilisé. C'est un algorithme fait pour les problèmes de

programmation discrète (Land and Doig, 1960). L'idée principale de cet algorithme est de représenter les cas de W sous forme d'arbre. Les nœuds enfants sont des partitions de leur parent. Les nœuds feuilles sont les composants de W. Notons que l'algorithme est exact et peut trouver la meilleure solution comme si on faisait le balayage de toutes les possibilités. L'objectif est de trouver la feuille avec la meilleure note dans l'arbre. L'algorithme d'origine calcule la valeur des nœuds intérieurs. Si le score S est plus grand que le maximum, il va être gardé. Sinon, le nœud et tous ces enfants vont être éliminé de l'algorithme. L'algorithme est présenté à l'Annexe D.

Pour pouvoir compléter l'algorithme, il faut choisir :

— la loi de sélection des nœuds : la façon d'évaluer les nœuds nous permet d'avoir des réponses qui correspondent à nos besoins.
— la structure de l'arbre : une bonne structure de l'arbre permet de bien éliminer les mauvaises solutions.
— l'algorithme pour calculer le score des nœuds : un bon algorithme permet d'évaluer avec précision la valeur des nœuds en maintenant le temps de calcul raisonnable.

Afin d'éliminer les couples d'observation/sous-carte qui ne sont pas similaires, *Cartographer* introduit dans l'algorithme un seuil de probabilité. À partir de ce seuil, les résultats sont acceptés, si le résultat ne dépasse pas ce seuil, cette solution ne nous intéresse pas. Puisque dans la pratique, ce seuil ne peut pas être dépassé régulièrement, le nombre de calculs est réduit dramatiquement. De plus, en utilisant l'algorithme de Depth First Search (DFS) [6] pour choisir les nœuds à évaluer, on évalue le nombre maximal de feuilles très rapidement et donc les meilleurs candidats sont évalués au début (s'il existe une solution optimale). La méthode complète est décrite à l'Annexe E.

Dans l'arbre, chaque nœud est caractérisé par un tuple $c = (c_x, c_y, c_\theta, c_h) \in Z^4$ où c_h est la hauteur du nœud dans l'arbre. Les nœuds de hauteur c_h peuvent présenter $2^{c_h} \times 2^{c_h}$ translations avec une rotation spécifique qui est $\bar{W}_c$:

$$\bar{W}_h = \left(\left\{ (j_x, j_y) \in Z^2 : \begin{array}{l} c_x \leq j_x < c_x + 2^{c_h} \\ c_y \leq j_y < c_y + 2^{c_h} \end{array} \right\} \times c_\theta \right) \tag{2.30}$$

$$\bar{W}_c = \bar{W}_h \cap \bar{W} \tag{2.31}$$

6. DFS commence depuis la racine de l'arbre et explore le plus loin possible (vers les feuilles) avant de revenir et essayer une autre branche (Tarjan, 1972)

Les feuilles sont alors des nœuds avec hauteur $h = 0$, ce qui correspond à la pose :

$$\xi_c = \xi_0 + (rc_x, rc_y, \delta_\theta c_\theta) \tag{2.32}$$

Dans la formule de l'Annexe E, l'ensemble des nœuds racines C_0 peut couvrir toute la région de recherche W avec une hauteur h_0.

$$\bar{W}_{0,x} = \{-w_x + 2^{h_0} j_x : j_x \in Z, 0 \leq 2^{h_0} j_x \leq 2w_x\} \tag{2.33}$$
$$\bar{W}_{0,y} = \{-w_y + 2^{h_0} j_y : j_y \in Z, 0 \leq 2^{h_0} j_y \leq 2w_y\} \tag{2.34}$$
$$\bar{W}_{0,\theta} = \{j_\theta \in Z, -w_\theta \leq j_\theta \leq w_\theta\} \tag{2.35}$$
$$C_0 = \bar{W}_{0,x} \times \bar{W}_{0,y} \times \bar{W}_{0,\theta} \tag{2.36}$$

Pour calculer la note des feuilles, on utilise la fonction à maximiser dans (2.26) :

$$\sum_{k=1}^{K} M_{nearest}(T_\xi h_k) \tag{2.37}$$

Ce qu'il reste est de trouver la façon efficace pour calculer le score des nœuds de hauteur $h \neq 0$. On utilise alors la formule suivante :

$$S(c) = \sum_{k=1}^{K} \max_{j \in \bar{W}_c} M_{nearest}(T_{\xi_j} h_k) \geq \max_{j \in \bar{W}_c} \sum_{k=1}^{K} M_{nearest}(T_{\xi_j} h_k) = S(c^* \in \bar{W}_c) \tag{2.38}$$

Pour calculer les nœuds de hauteurs $h \neq 0$, *Cartographer* utilise des grilles pré-calculées pour chaque hauteur h. Les grilles pré-calculées M^h_{pre} ont la même dimension que $M_{nearest}$, la différence est que chaque pixel de la grille pré-calculée contient la valeur maximale de $2^h \times 2^h$ pixels autour de lui dans la grille d'origine $M_{nearest}$ comme illustré à la figure 2.8. La note des nœuds qui ne sont pas des feuilles est donné par :

$$S(c) = \sum_{k=1}^{K} M^{c_h}_{pre}(T_{\xi_c} h_k) \tag{2.39}$$
$$M^h_{pre}(x, y) = max_{(x',y') \in \bar{W}_h} M_{nearest}(x', y') \tag{2.40}$$

Les grilles pré-calculées expliquent le fait que l'on doit attendre que les sous-cartes soient complètes avant de faire l'optimisation. Si les grilles de probabilités de la carte ne varient plus, nous devons calculer les grilles pré-calculées seulement une fois.

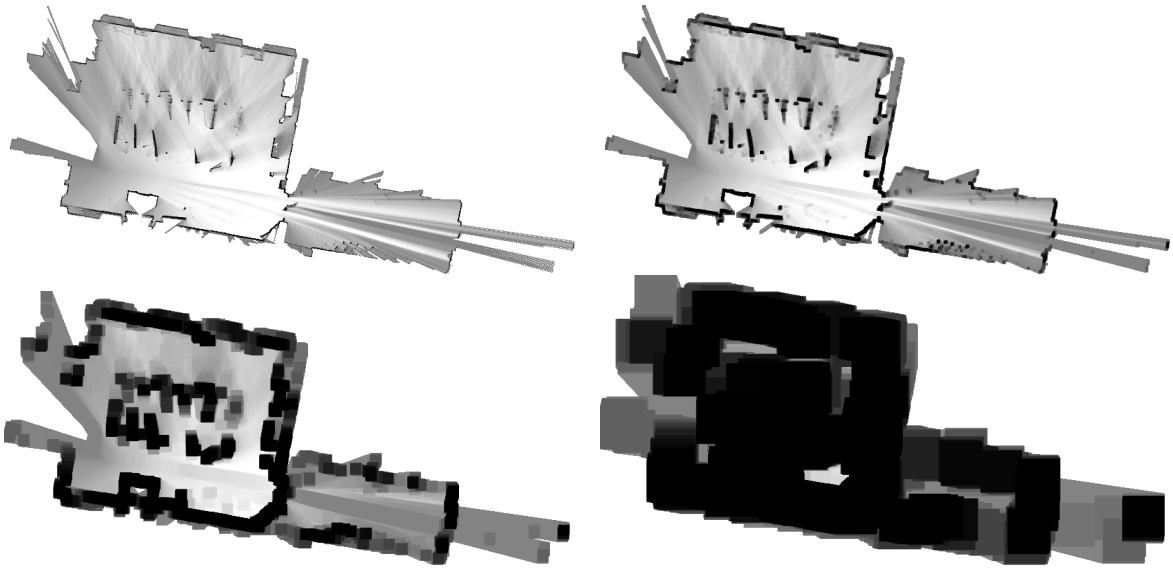


Figure 2.8 Grilles pré-calculées de hauteur 1,4,16 et 64 (Hess et al., 2016).

2.3 Comparaison des algorithmes de SLAM

2.3.1 Plateforme de test des algorithmes

Dans cette section, les expériences faites pour comparer la performance des trois algorithmes de SLAM sont présentées. L'évaluation sera faite en simulation dans un environnement créé dans *Gazebo*. En créant nous-mêmes l'environnement, on peut connaître la carte exacte de l'environnement (*ground truth*).

L'environnement est créé avec les caractéristiques qui peuvent poser des problèmes pour les algorithmes de SLAM, par exemple, les longs couloirs, les salles identiques, la fermeture des boucles, etc. C'est une grande salle de $20 \times 20\,\text{m}$ qui est visualisé en 3D avec la carte exacte en 2D à la figure 2.10.

Pour évaluer la performance des algorithmes, nous allons utiliser l'algorithme ICP pour comparer la carte créée par le robot et la carte exacte de l'environnement. ICP est souvent utilisé pour reconstruire les images 2D et 3D à partir de plusieurs observations. Il fut introduit la première fois par Chen and Medioni (1991); Besl et al. (1992). La technique compare les nuages de points (*point cloud*) en cherchant une configuration qui minimise la distance entre les 2 nuages avec les 4 étapes ci-dessous et suivant la figure 2.11

1. Pour chaque point dans le nuage de début, associer un point dans l'autre.
2. Enlever les points aberrants.
3. Estimer la transformation optimale qui minimise la distance totale point par point

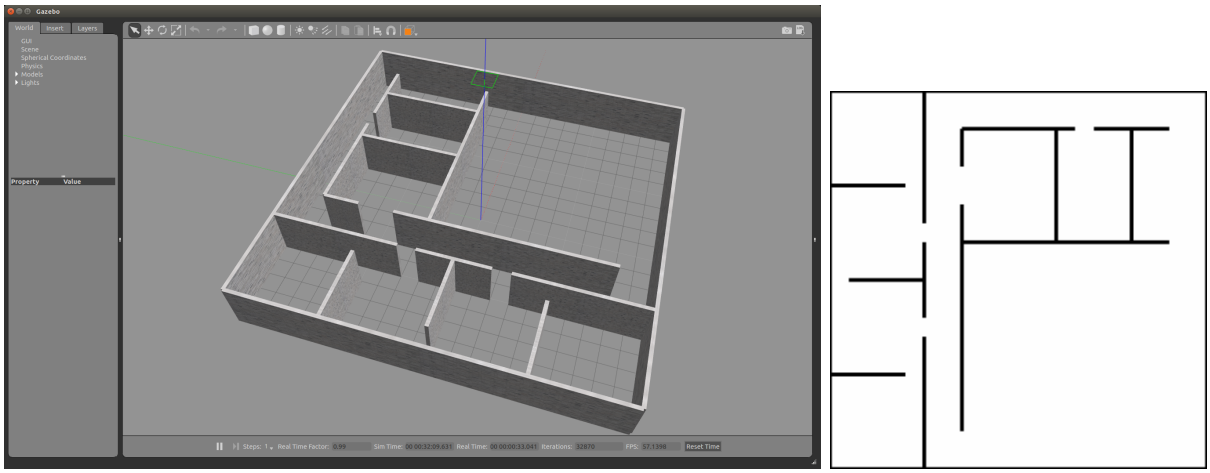


Figure 2.10 Environnement de test en 3D dans *Gazebo* et sa grille d'occupation en 2D

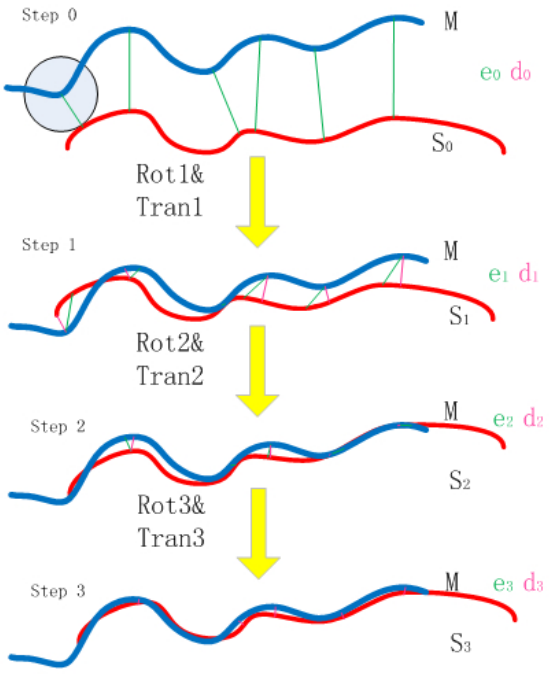


Figure 2.11 Étapes d'une itération de l'algorithmes ICP

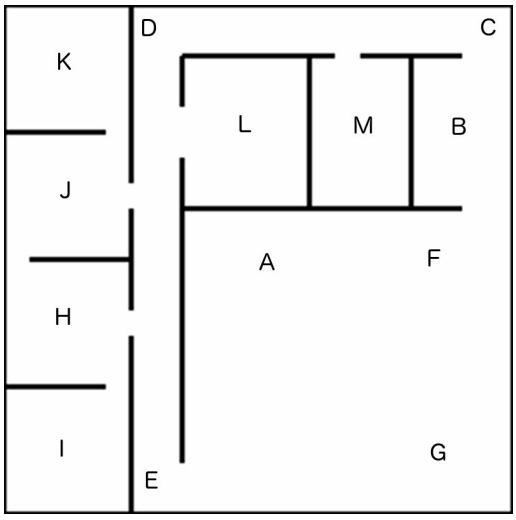


Figure 2.12 Trajectoire de test du robot

des deux nuages.

4. Ré-associer les points jusqu'à ce qu'on trouve la configuration optimale (ou dépasse le seuil souhaité).

Une évaluation de la charge de calcul de l'ordinateur est faite avec celle de la précision de la carte créée par chaque algorithme en la comparant avec la carte exacte. La charge du processus est enregistrée durant la procédure de cartographie. Les données utiles sont seulement les données quand le processus de cartographie est en marche (*running*). L'évaluation des cartes est faite sous *Matlab* en utilisant le module *Computer Vision System Toolbox* pour transformer une grille d'occupation en nuage de points et exécuter l'algorithme ICP. Pour tous les algorithmes de SLAM utilisés, la carte finale est construite avec une résolution de 0.05 m/pixel et la portée utile du LIDAR est de 7 m. La trajectoire du robot est selon la figure 2.12 :

— La position initiale du robot est au point A (le centre de la carte).
— Le robot longe le mur, passe le point F et entre dans la salle B.
— Il sort de la salle B, retrouve le point C et suit le long mur passant par les points D et puis E.
— Il entre dans la grande salle et fait un tour autour de la salle, passant par les points A, F et puis G.
— Il sort de la salle au point E, entre dans les petites salles suivant l'ordre $H \to I \to H \to J \to K \to J$.
— Il sort des petites salles à gauches et entre dans la salle L.
— Il sort de la salle L, passe par le coin D afin d'entrer dans la salle M et termine sa mission d'exploration.

2.3.2 Résultats

En faisant les simulations, on constate que les trois algorithmes sont capables de livrer une bonne carte globale. La figure 2.13 montre la charge de calcul de l'ordinateur en fonction du temps pour les trois algorithmes. Avec la figure 2.13 et le tableau 2.2, on peut constater que :

— Le meilleur algorithme en fonction de la charge moyenne est *Cartographer* qui est loin des deux autres algorithmes. La pire performance est *GMapping* qui utilise en moyenne presque 80% de la capacité de calcul de l'ordinateur.
— À cause de la grande charge de calculs, *GMapping* ne peut pas donner une carte en temps réel comme les deux autres algorithmes (il existe toujours un délai d'environ $1 - 2$ s dans la procédure de cartographie de *GMapping*).

— En revanche, *Cartographer* est l'algorithme qui a le plus grand écart-type. Ses excursions sont créées périodiquement quand il essaye de fermer la boucle.
— Au début du programme, il n'y a pas assez de sous-cartes complètes pour faire la fermeture de boucle, *Cartographer* montre une meilleure performance.

Tableau 2.2 Charge de l'ordinateur des trois algorithmes (Moyenne ($\bar{x}$), médiane ($\tilde{x}$), écart-type (σ))

La charge des trois algorithmes			
	GMapping	KartoSLAM	GoogleCartographer
$\bar{x}$	78.2 %	69.8 %	49.7 %
$\tilde{x}$	80.0 %	80.0 %	26.7 %
σ	24.4 %	29.0 %	37.5 %

La figure 2.15 montre les trois cartes créées par les algorithmes. Les trois cartes sont enregistrées en utilisant le module *map_server* de ROS. On peut voir facilement que la qualité de *GMapping* est moins bonne que celles des autres, du fait de la précision de la fermeture des boucles, quand on ferme la boucle dans *GMapping*, c'est la particule la plus probable qui est choisie pour fermer la boucle et corriger la trajectoire du robot. Le filtre particulaire de *GMapping* peut maintenir au maximum 30 particules (donc 30 trajectoires). La solution optimale peut être alors rejetée durant le processus si le robot fait une longue trajectoire avant de revenir à la place précédente. *KartoSLAM* donne une meilleure solution. La fermeture de boucle est bien faite car l'arbre des poses de *Karto* peut maintenir toutes les positions du robot et tout prendre en compte dans les calculs. L'algorithme qui donne la meilleure carte est *Cartographer*. En revanche, on peut voir des zones inconnues dans la carte parce que la carte de *Cartographer* est une grille de probabilités. Le module *map_server* ne nous donne qu'une carte avec trois états des pixels (inconnu, libre et occupé). Les seuils de probabilités du module *map_server* sont : 0.65 pour un pixel occupé et 0.196 pour un pixel libre. Les pixels dont la probabilité est au milieu des deux seuils sont considérés inconnus dans la carte ; c'est pourquoi la zone inconnue de *Cartographer* est plus grande. Les cartes des deux algorithmes *KartoSLAM* et *Cartographer* sont faites en temps réel et la procédures de fermeture des boucles a un léger délai ($1 - 2\,\mathrm{s}$) ce qui donne une impression semi-temps réel à l'utilisateur. En plus de la fermeture des boucles, la façon de maintenir deux grilles de probabilités pour les pixels *hits* et *misses* donne une meilleure performance vis-à-vis du problème d'obstacles dynamiques (les gens, les animaux, etc.) et semi-dynamiques (les chaises, les bureaux, etc.).

La figure 2.16 donne une comparaison des cartes créées avec la carte exacte de l'environnement (*ground truth*). Les nuages des points sont comparés avec le module *Computer Vision System Toolbox* de *Matlab*. La précision des algorithmes est confirmée encore une

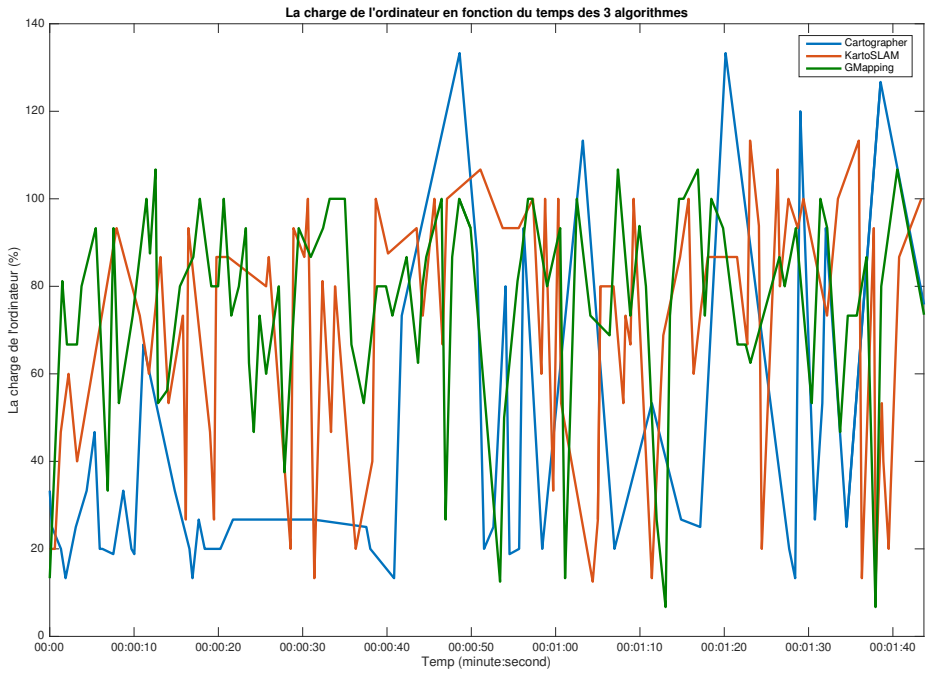
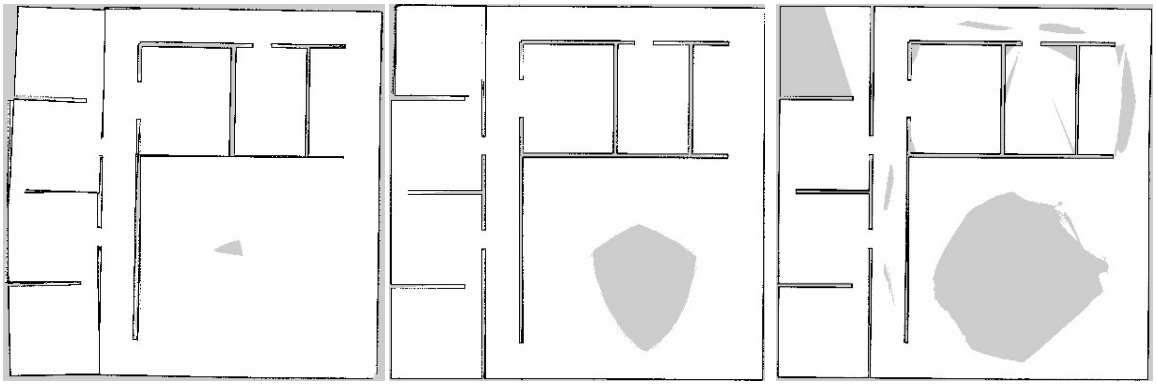


Figure 2.13 Charge de l'ordinateur en fonctions du temps des trois algorithmes

fois. Avec l'algorithme de comparaison ICP, on peut avoir aussi la moyenne des distances euclidiennes des paires de points après la comparaison, ce qui peut représenter l'erreur entre les deux cartes. Pour *GMapping*, l'erreur est de 9.25 cm/point. L'erreur de *KartoSLAM* est 7.58 cm/point, et celle de *Cartographer* est la plus petite, 6.17 cm/point.

2.3.3 Conclusion

En conclusion, notre expérience montre la robustesse des techniques SLAM basées sur la représentation par un graphe car c'est une méthode qui considère toute la trajectoire du robot et toutes les données du LIDAR. *GMapping* donne à la sortie une carte qui est bonne mais qui n'est pas suffisante pour notre application. En outre, la charge de l'ordinateur avec *GMapping* est la plus grande, ce qui n'est pas recherché. On va vouloir inclure la carte *a priori* dans notre algorithme, celle-ci n'est pas explicitement présente dans les divers techniques de SLAM 2D actuellement. En plus, on va fermer beaucoup de boucles dans l'exploration, car presque tous les points dans la trajectoire existent déjà dans la carte *a priori*. Donc, on favorise un algorithme qui maintient le poids de la carte *a priori* et efficace au niveau des fermetures des boucles. La meilleure méthode pour notre cas est alors *Cartographer*, en se basant sur la représentation par un graphe, elle peut maintenir une carte dans un grand environnement.



(a) GMapping (b) KartoSLAM (c) Cartographer

Figure 2.15 Cartes créées par les algorithmes de SLAM

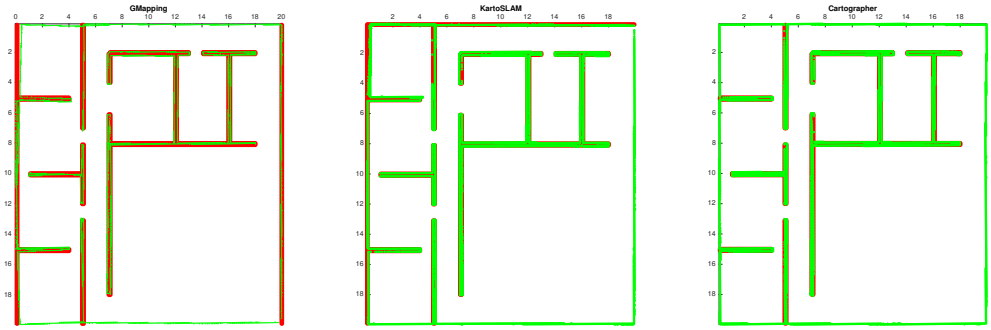


Figure 2.16 Comparaison des cartes créées (verte) avec la carte exacte de l'environnement (rouge)

Et en utilisant la bonne procédure d'optimisation, elle peut faire la fermeture de boucle en utilisant moins de puissance de calcul. Pour notre application, on veut charger une carte de l'environnement dans l'algorithme de SLAM et comparer les nouvelles données à cette carte. Donc la puissance et la vitesse de calcul pendant la fermeture des boucles est très importante. La performance vis-à-vis le problème d'obstacles dynamiques et semi-dynamiques est aussi un point fort de *Cartographer*.

Pour la suite du projet, nous utilisons le module *Cartographer* pour faire le SLAM de notre fauteuil. Le module de chargement de carte préliminaire sera fait en se basant sur *Cartographer*.

CHAPITRE 3 SLAM DANS UN ENVIRONNEMENT PARTIELLEMENT CONNU

3.1 Introduction

Pour pouvoir naviguer avec la commande collaborative, nous devons préciser une ou des destinations sur la carte de l'environnement. Nous devons donc avoir la carte dans le programme avant de faire la navigation collaborative. Avec les algorithmes de SLAM 2D actuels, cette tâche nous demande de naviguer manuellement et de faire la carte globale avant toute autre activité. Ce chapitre est consacré à la présentation d'une méthode permettant de charger une trajectoire artificielle/carte *a priori* dans le programme avec seulement les données essentielles et statiques de l'environnement (soit la carte architecturale de l'environnement, soit une carte faite par le SLAM). Nous présentons d'abord les besoins de notre système, la raison qui nous mène au développement d'un tel module et l'architecture de la solution envisagée dans la section 3.2. La section 3.3 présente l'algorithme proposé. Enfin, la dernière section discute les résultats obtenus avec cette méthode.

3.2 Solution envisagée

3.2.1 Besoins du système

Commande collaborative

Le but du projet est de perfectionner la commande collaborative et ses activités auxiliaires pour garantir l'autonomie du système et améliorer le confort des utilisateurs. Le module de

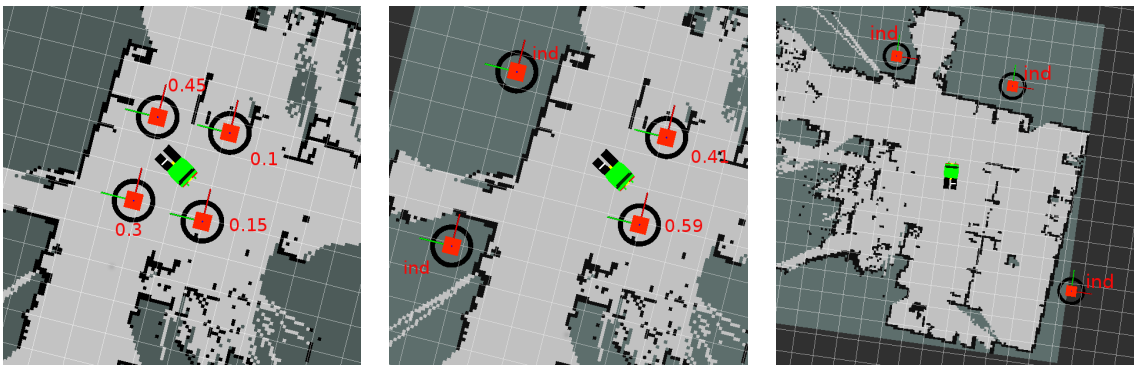


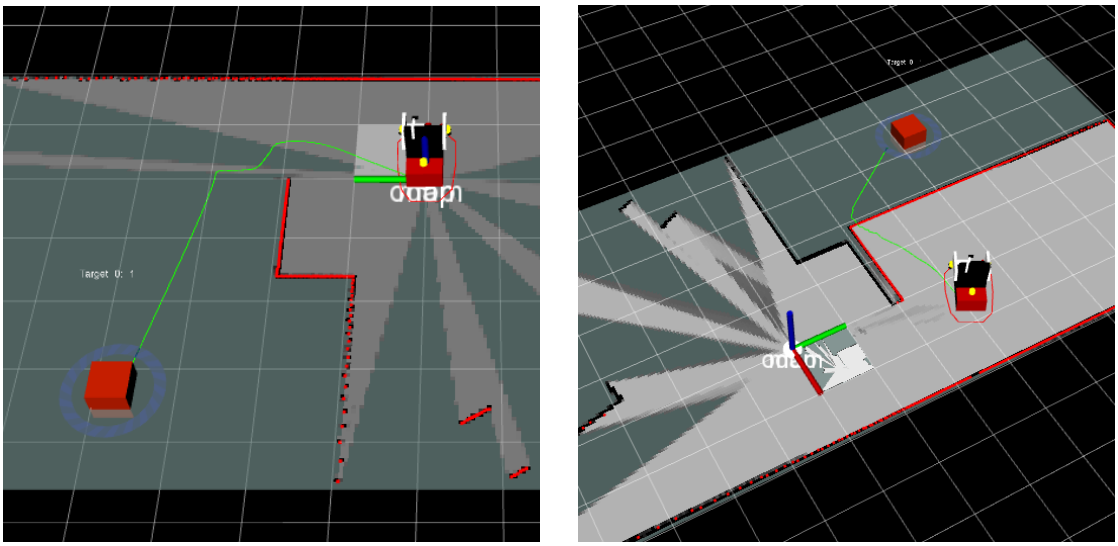
Figure 3.2 Estimation de destination de l'algorithme de commande (Ghorbel, 2016)

commande collaborative de Ghorbel (2016) assure les points suivant :

1. Estimation de l'intention de l'opérateur en se basant sur la carte de l'environnement et sur les commandes du *joystick* pour prédire la destination finale la plus probable dans l'ensemble des destinations prédéfinies. Cet ensemble est fixé et mis à jour par l'utilisateur.
2. Correction des commandes de l'opérateur pour atteindre la destination en combinant la commande autonome et la commande manuelle de l'utilisateur.
3. Évitement de collision pour la sécurité de l'utilisateur en cas de malfonctionnement du système ou en cas de la négligence de l'utilisateur.

Afin de réaliser le premier point, selon Ghorbel (2016), les destinations en dehors de la carte sont considérées inaccessibles et ne sont pas prises en compte par l'algorithme de commande (Fig. 3.2). Il est alors indispensable d'avoir une carte de l'environnement dans le programme afin de bien utiliser la commande collaborative. Avec les algorithmes de SLAM traditionnels, nous devons explorer l'environnement avant d'obtenir la carte, et avec les algorithmes de localisation dans un endroit avec la carte *a priori*, nous n'avons pas les données les plus récentes de l'environnement. Ces solutions ne sont pas efficaces. Nous allons concevoir un module qui charge la carte dans l'algorithme de SLAM choisi. En outre, la tâche de déterminer les destinations prédéfinies ne sont pas évidente pour les utilisateurs typiques du fauteuil. Un module de placement automatiquement les destinations est donc présenté au chapitre 4.

Le robot commence le trajet avec *Cartographer* dans un environnement inconnu comme illustré à la figure 3.4a. Dans le plan de l'environnement de test, c'est le point D illustré à la figure 3.15, l'on veut que la destination finale soit dans la salle E dans la carte. On peut constater que si on a une seule destination sur la carte, la probabilité de la destination est 1, et la trajectoire est toujours vers cette destination. Mais comme le robot ne possède pas une carte de l'environnement *a priori*, la trajectoire planifiée passe par la salle C. Dans ce cas, le robot rentre dans une impasse dont la seule solution est de faire un détour et chercher une autre trajectoire (par exemple, traverser le couloir). Dans notre cas illustré à la figure 3.4b, le robot n'est même pas capable de trouver une autre trajectoire dans la carte qui l'amène à la destination une fois dans la salle C. Ceci est dû à la taille de la carte limitée aux observations du robot. Avec seulement ces données, il n'y a aucune trajectoire disponible. Donc, pour rejoindre le deuxième point, il est aussi nécessaire d'avoir une carte *a priori* de l'environnement.



(a) (b)

Figure 3.4 Trajectoire planifiée et sa carte de l'environnement

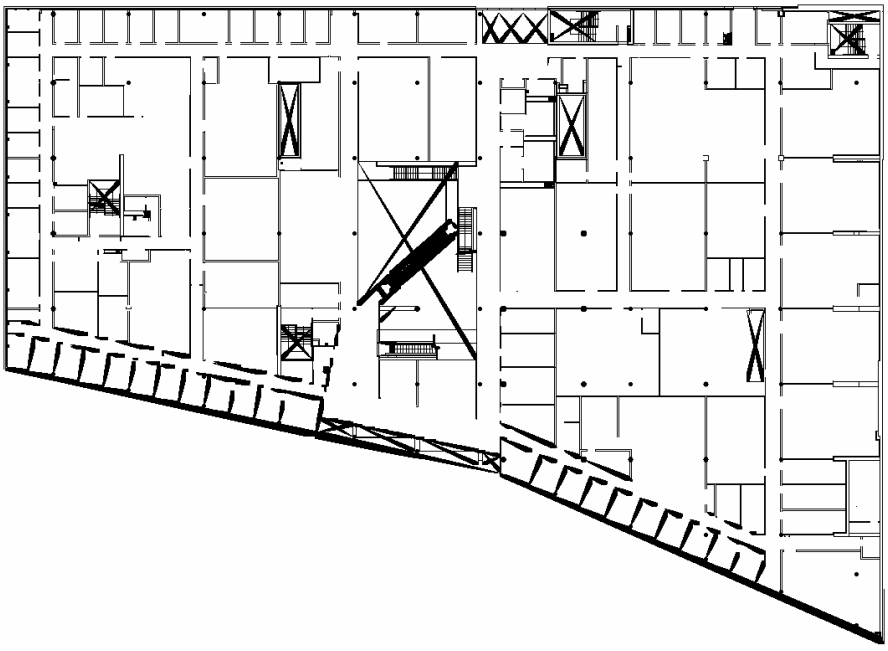
Construction des trajectoires virtuelles à partir de cartes architecturales

Afin de charger la carte dans le programme pour l'utiliser, nous devons premièrement obtenir une carte utilisable par notre système. Afin d'utiliser les cartes architecturales, Majdoub (2014) a proposé une méthode pour les transformer sous forme de grilles d'occupation. La technique prend en entrée une carte architecturale sous format Drawing eXchange Format (DXF) (format pour les dessins techniques dont la référence est fournie par Autodesk). L'entrée de l'algorithme est illustrée dans la figure 3.6a. À la sortie, l'algorithme de Majdoub (2014) nous fournit la grille d'occupation de l'environnement. Le résultat est illustré à la figure 3.6b dans laquelle les régions fermées sont marquées par un $\times$. Un dernier traitement est fait pour rendre la carte compatible pour notre programme. Il faut marquer la région en dehors de l'étage comme inconnu. Pour réaliser la tâche, nous suivons les étapes suivantes :

— Convertir la carte en une carte binaire pour faire les traitements d'images.
— Chercher les contours dans l'image.
— Chercher le contour de taille maximale. On obtient un polygone entourant la carte.
— Faire un masque de l'image avec ce polygone.
— Dilater le masque pour que le masque contienne les murs dans la carte et pour enlever les bruits.
— Marquer comme inconnus les points dans la carte originale dont les coordonnées sont en dehors du masque.

La carte obtenue est illustrée à la figure 3.7. C'est une carte qui est utilisable par notre

(a) Carte architecturale sous format DXF



(b) Résultat de l'algorithme de Majdoub (2014)

Figure 3.6 Construction de grille d'occupation à partir de le carte architecturale

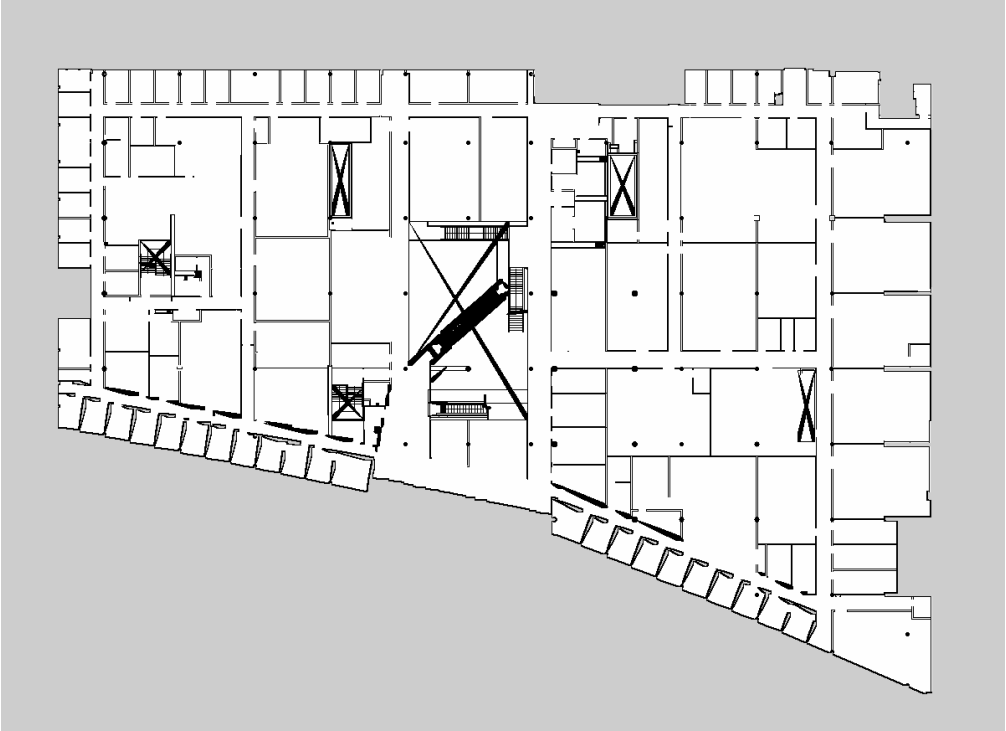


Figure 3.7 Carte utile pour notre programme

programme. Si nous ne faisons pas ce traitement, l'algorithme de génération de trajectoire de la section 3.3 et l'algorithme de segmentation de salle du chapitre 4 va donner de mauvais résultats car ils vont prendre en compte les régions inconnues de la carte et les traiter comme des régions libres.

Chargement de carte dans l'algorithme de SLAM

Nous voulons charger une carte simplifiée de l'environnement dans l'algorithme de SLAM. Pour pouvoir réaliser la tâche, nous devons connaître la position actuelle du robot dans la carte (car au début du SLAM, la position du robot doit coïncider avec l'origine de la carte). Le problème peut se traduire en un problème de SLAM de plusieurs robots sans connaître la position relative initiale des robots et sans connaître le moment de rendez-vous de deux des robots. Certains travaux tentent de résoudre le problème en connaissant le moment de rendez-vous des robots (Howard, 2004; Zhou and Roumeliotis, 2006; Andersson and Nygards, 2008; Carlone et al., 2010). Ces articles utilisent l'hypothèse que les robots se reconnaissent et profitent du moment de rendez-vous pour améliorer la localisation globale du système. Il existe aussi des algorithmes pour résoudre le problème complet sans connaître la position relative des robots (Franchi et al., 2013; Indelman et al., 2014; Saeedi et al., 2015). Par

contre, les méthodes précédentes dépendent fortement de la ressemblance des observations. Dans le cas où l'on veut utiliser des cartes architecturales, la tâche d'apparier les nouvelles observations avec l'ancienne carte va échouer. L'option qui reste pour ce problème est de localiser le robot dans la carte a priori avant de la charger dans l'algorithme de SLAM. Pour cela, nous utilisons le module Adaptive Monte Carlo Localization (AMCL) de ROS. AMCL est un algorithme de localisation basé sur le filtre de particules. Il utilise l'approche d'échantillonnage adaptatif (Kullback–Leibler divergence (KLD)) proposé par Thrun et al. (2005) pour réduire le nombre de particules dans l'algorithme et pour réduire le problème de duplication de particules.

Les algorithmes de SLAM 2D peuvent, en théorie sauvegarder et recharger les cartes. Pour les algorithmes basés sur le filtre particulaire, il est nécessaire de sauvegarder toutes les particules générées, et pour les algorithmes basés sur la représentation par un graphe, une sauvegarde de l'arbre des poses suffit. Il reste maintenant à générer un arbre de poses et des observations artificielles à partir de la carte de l'environnement. On utilise ensuite le graphe de Voronoi pour obtenir les poses de la trajectoire.

3.2.2 Structure proposée

La structure du programme est décrite à la figure 3.8. Au début du programme, on utilise AMCL pour déterminer la position du robot dans la carte de l'environnement. Notons que cette position doit être sauvegardée dans *Cartographer* pour l'utiliser ultérieurement dans l'algorithme de la section 4.3.2 qui permet de gérer des destinations. Ensuite, la carte est utilisée pour générer la trajectoire et les observations artificielles du robot. La méthode est présentée dans la section 3.3. Toutes ces données sont ensuite transférées à l'algorithme de *Cartographer* pour former la trajectoire du premier robot. Nous utilisons une transformation

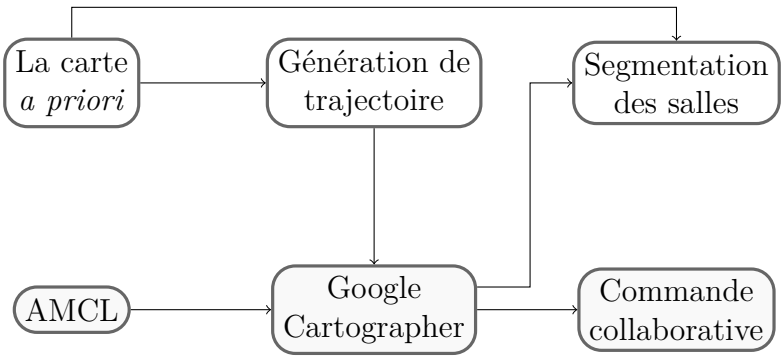


Figure 3.8 Structure du SLAM amélioré

de repère pour placer le robot à l'origine de la nouvelle carte créée. L'amélioration donne à l'utilisateur une vue globale de l'environnement à proximité du robot et donne au robot la capacité de planifier la trajectoire globale vers une destination qui n'a pas encore été visitée.

3.3 Génération de la trajectoire virtuelle

3.3.1 Graphe de Voronoi

Le diagramme de Voronoi est un outil qui est utilisé dans plusieurs domaines des sciences et de la technologie. Avec un ensemble de sites prédéfinis $\{p_1, ..., p_n\}$, le diagramme divise l'espace X en plusieurs cellules R_i définies comme suit :

$$R_k = \{x \in X | d(x, p_k) \leq d(x, p_j) \quad \forall j \neq k\} \tag{3.1}$$

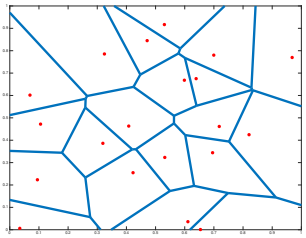


Figure 3.9 Exemple du diagramme de Voronoi

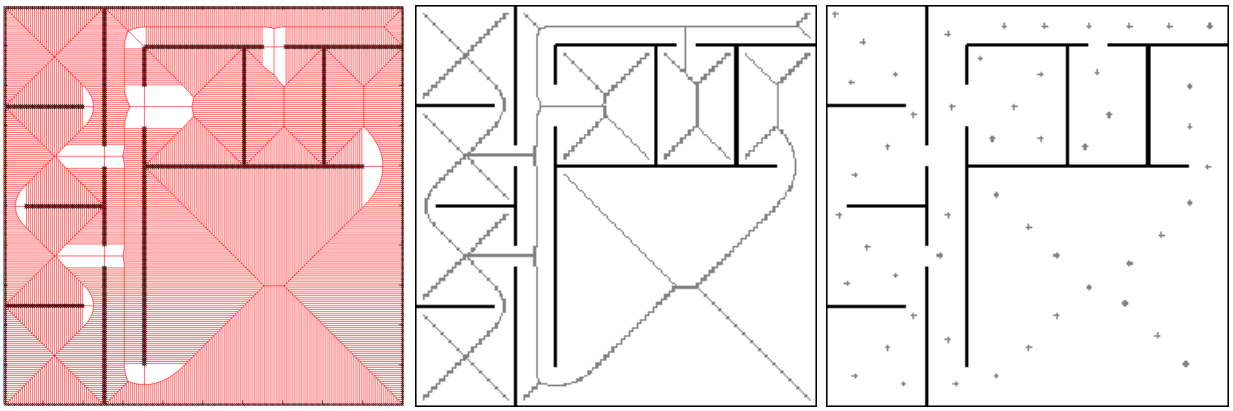
Tous les points dans la région R_i sont plus proches de p_i que tous les autres sites dans l'espace. La figure 3.9 montre un exemple simple du diagramme avec un ensemble de vingt sites qui sont aléatoires et sont marqués en rouges. Les segments en bleu dans la figure sont les frontières des régions R_i : ils représentent l'ensemble des points dans l'espace qui sont à la même distance des sites les plus proches.

Dans le domaine de la robotique, et plus spécifiquement la navigation autonome, on utilise le diagramme de Voronoi pour chercher des trajectoires dans l'environnement. On génère le diagramme de Voronoi à partir d'une carte, et les sites sont les obstacles dans cette carte. Les frontières dans le diagramme sont les trajectoires possibles du robot dans la carte. Les frontières de Voronoi sont placées le plus loin possible des obstacles.

3.3.2 Génération des données du capteur

Dans notre application, on veut générer une trajectoire virtuelle du robot qui doit couvrir toutes les régions accessibles et qui n'est pas trop lourde en termes de temps de calcul. Donc la procédure pour réduire le diagramme de Voronoi est la suivante :

— Puisque dans une carte, on a plusieurs points occupés côte à côte. Le diagramme de Voronoi est donc très complexe comme illustré à la figure 3.11a. On veut alors enlever les frontières qui sont proches des obstacles. On fait une dilatation des obstacles de la carte pour ensuite effacer toutes les frontières qui touchent ces obstacles dilatés. Le



(a) (b) (c)

Figure 3.11 Trajectoire virtuelle du robot

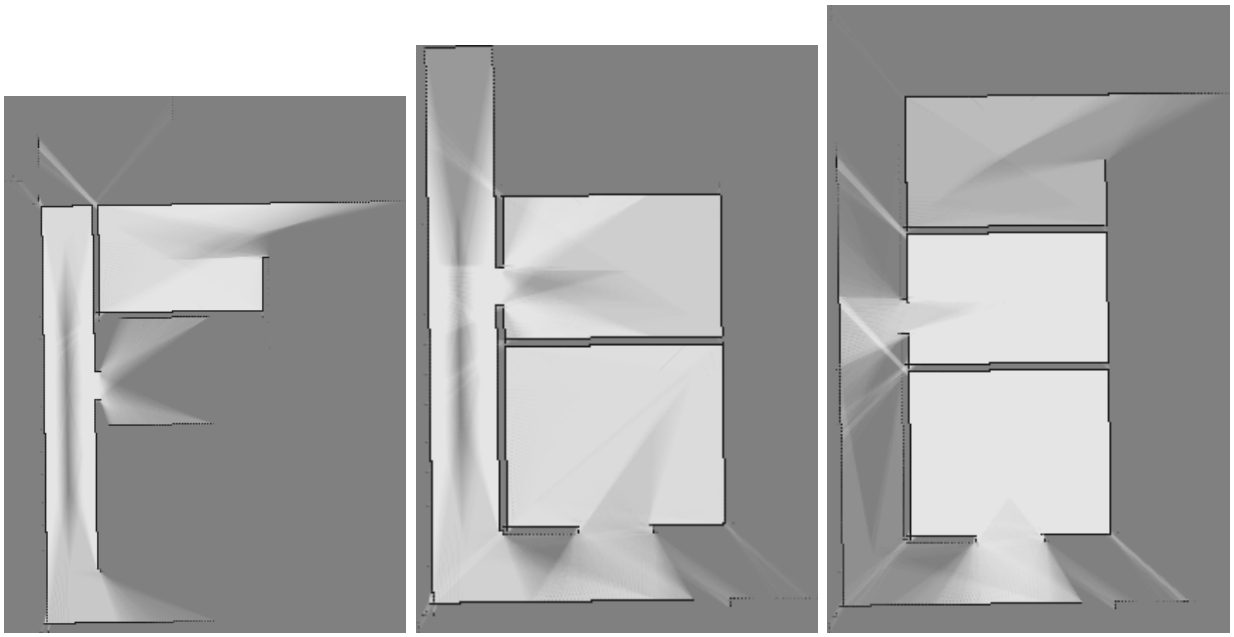
résultat obtenu est illustré à la figure 3.11b

— On cherche ensuite les nœuds qui n'ont pas de voisins dans un certain rayon (environ 2 m) et on enlève les points dont la distance au site le plus proche est trop petite. On veut limiter le nombre de données dans la trajectoire virtuelle et enlever les régions non accessibles par le robot. Le résultat obtenu après ces deux étapes est illustré à la figure 3.11c

La réduction de la carte de l'environnement de test de la figure 3.15 est faite avec la procédure précédente comme illustré dans la figure 3.11. Pour générer les données du capteur, nous plaçons un LIDAR virtuel dans les positions trouvées et faisons un balayage autour du LIDAR jusqu'à un rayon désiré. Les données sont sauvegardées dans le format requis par l'algorithme de SLAM (par exemple, *KartoSLAM* utilise un tableau de distances avec l'angle de mesure qui incrémente, et *Cartographer* utilise un tableau de points dont les coordonnées sont dans le repère du LIDAR).

3.3.3 Chargement de la trajectoire dans *Cartographer*

Pour charger la trajectoire dans *Cartographer*, on tente d'abord de trier les points de la trajectoire afin de minimiser la distance totale pour se déplacer à tous ces points. Ceci représentente un problème du voyageur de commerce (*travelling salesman*). Résoudre ce problème de façon exacte surcharge la charge de calcul dans le système, car le nombre de cas possibles est $\frac{1}{2}(n-1)!$. Nous prenons donc un algorithme simple pour le résoudre, i.e., l'algorithme glouton (*greedy algorithm*), qui déplace le fauteuil vers le point plus proche à chaque itération. L'algorithme va générer une trajectoire qui est de l'ordre de grandeur de $O(\log n)$ plus longue que la trajectoire optimale d'après Johnson and McGeoch (1997). Si



(a) (b) (c)

Figure 3.13 Sous-cartes *a priori* du robot

la distance totale est petite, on peut obtenir des sous-cartes qui sont plus précises car elles possèdent plusieurs observations dans le même endroit. La figure 3.13 nous montre trois exemples de sous-cartes consécutives. On peut voir qu'avec le triage, les sous-cartes ont une bonne précision dans une petite région, ce qui va nous aider quand on veut apparier les sous-cartes entre elles et quand on veut les apparier avec les observations du robot.

La trajectoire est ensuite chargée dans *Cartographer* et elle est marquée comme la trajectoire d'un autre robot. Par conséquent, la trajectoire est utilisée dans les fermetures des boucles pour améliorer la précision globale de la carte créée par le SLAM. Comme les cartes *a priori* sont plus simples que les cartes construites par des algorithmes de SLAM, il est plus facile de faire une fausse fermeture de boucle. Donc, on veut augmenter la note minimale pour la fermeture des boucles globales à environ 0.8.

3.4 Expérimentation

3.4.1 Environnement de test

La figure 3.15 montre notre environnement de test en simulation. On ajoute des obstacles dans l'environnement pour simuler la dynamique du milieu. Le robot part du point A, il sort de la salle et suit le couloir jusqu'au point B. Il retourne dans le couloir et entre dans la

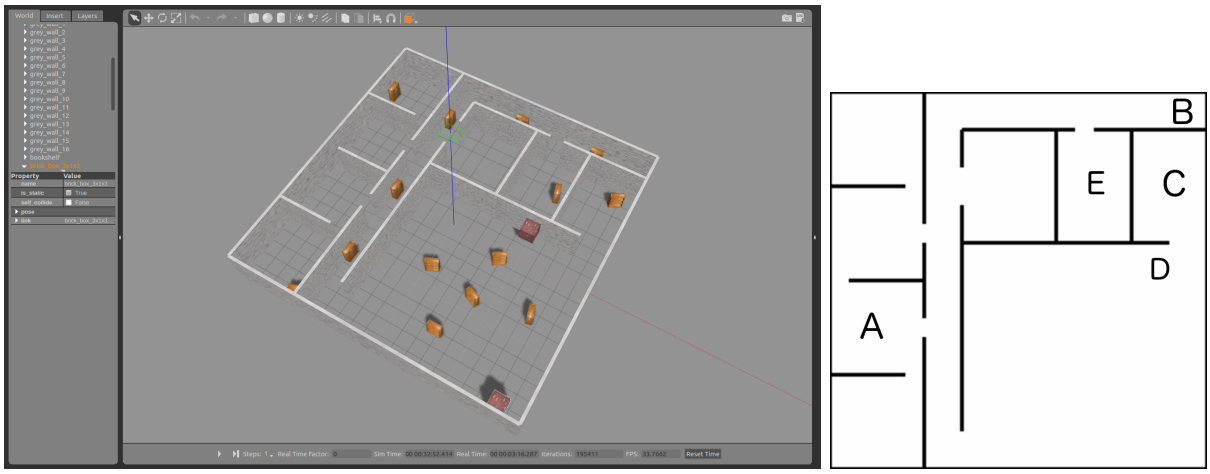


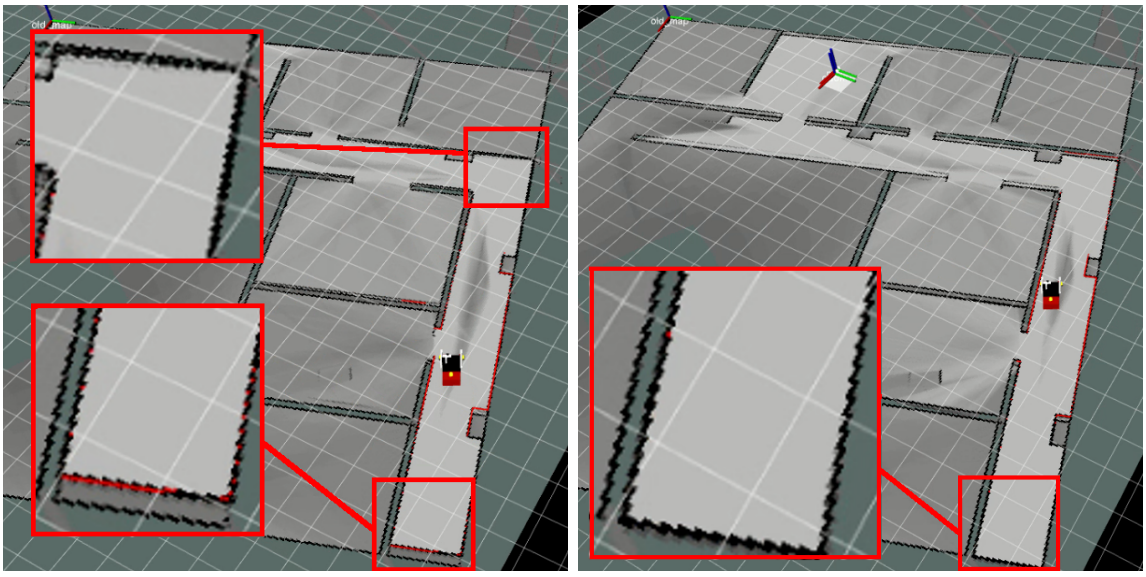
Figure 3.15 Environnement de test en 3D dans *Gazebo* et sa grille occupation en 2D

grande salle avant d'entrer dans la salle C qui est la destination finale. Avec cette trajectoire, on veut illustrer la différence dans la fermeture des boucles pour ces deux cas.

3.4.2 Résultats

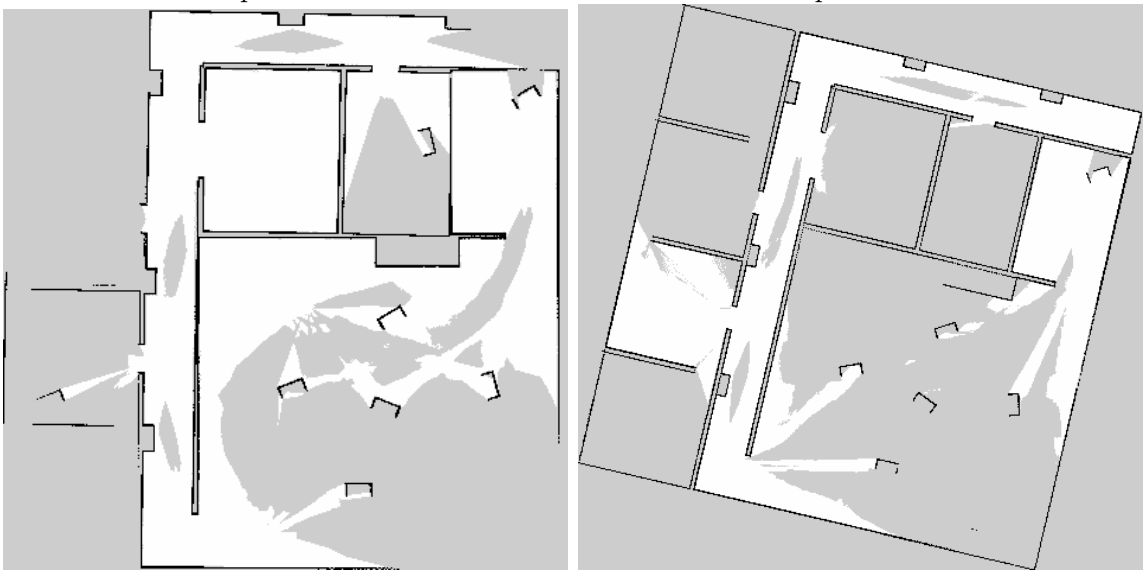
La figure 3.17 montre les cartes faites avec *Cartographer* avec et sans la carte *a priori*. La figure 3.17c est faite par l'algorithme original de *Google*. On peut voir que la fermeture de boucle ne prend pas en compte l'épaisseur du mur et combine alors les deux côtés du mur ensemble. De plus, la carte est tournée par rapport à la position d'origine du robot dont l'orientation est 0^o.

La figure 3.17d est la carte créée après avoir chargé la carte de l'environnement. On peut voir clairement que la qualité de la carte est meilleure. Les séparations des salles et des couloirs sont plus exactes dans ce cas. En effet, au début de la trajectoire, le robot a un peu dévié, ce qui a pour effet que le mur du couloir et de la salle se mettent ensemble et que le couloir devient plus court comme dans la figure 3.17a. Cette déviation est due aux incertitudes du LIDAR ou de l'odométrie du robot. Avec la fermeture de boucle, *Cartographer* a apparié le couloir de la carte prédéfinie et les observations comme illustré à la figure 3.17b. Il améliore la carte et corrige les erreurs dues au manque d'informations afin d'obtenir la carte finale qui est meilleure comme illustré à la figure 3.17c et 3.17d. Cela a augmenté la qualité de la fermeture des boucles dans ce cas où on n'a pas assez d'informations. On voit aussi que les obstacles sont facilement ajoutés dans la nouvelle carte car la probabilité des espaces libres choisis est basse dans la carte initiale (la couleur gris foncé dans les figures 3.17a et 3.17b).



(a) Carte avant la fermeture de boucle qui utilise la carte *a priori*

(b) Carte après la fermeture de boucle qui utilise la carte *a priori*



(c) Carte finale sans les données *a priori*

(d) Carte finale avec les données *a priori*

Figure 3.17 Carte créée par *Cartographer* avant et après insertion de la carte architecturale

3.4.3 Conclusion

En conclusion, l'algorithme s'adapte bien à notre besoin, car une bonne cartographie est indispensable pour la navigation. En effet, avec l'algorithme original de *Cartographer*, nous pouvons aussi obtenir une bonne carte comme dans la figure 2.15c. Par contre, la trajectoire doit être plus complexe pour assurer la fermeture des boucles. Si on est trop loin de l'origine, les erreurs peuvent s'accumuler et dépasser la taille de la fenêtre de recherche. Avec la trajectoire simple, même en accumulant de petites erreurs, si on navigue dans un grand environnement (par exemple une école, un centre commercial, etc.), elles peuvent devenir significatives. Cela peut nous amener à une mauvaise destination avec la commande collaborative. Avec le changement, on peut avoir une carte précise avec une trajectoire simple. L'utilisateur peut alors utiliser directement la commande collaborative avec SLAM sans pré-établir la carte complète de l'environnement.

CHAPITRE 4 SEGMENTATION DES SALLES DANS LA CARTE

4.1 Introduction

Dans le chapitre précédent, nous avons proposé une méthode pour obtenir une carte initiale dans le SLAM sans naviguer dans l'environnement. Dans ce chapitre, nous proposons une méthode pour choisir des destinations potentielles et les mettre dans la carte. La section 4.2 présente une revue de littérature sur les méthodes de segmentations des salles à partir de la carte architecturale qui nous permet de choisir une méthode convenable à implémenter dans le programme. La deuxième partie, section 4.3, est la présentation du module de mise en œuvre et la nouvelle structure de la commande collaborative adaptée à un tel module. Enfin, la section 4.4 discute des résultats du programme final implémenté sur le fauteuil.

4.2 Revue de littérature

L'idée principale de la segmentation des salles est que les usagers veulent naviguer dans les couloirs, et la destination finale de la trajectoire est toujours une salle. Dans ce cas-là, pour trouver les destinations possibles sur la carte, on tente de déterminer les éléments qui sont des salles. Ce problème est souvent appelé la segmentation des salles (*room segmentation*). Depuis plusieurs années, le problème de division de la carte architecturale en plusieurs salles (ou régions importantes) est le cœur de plusieurs projets dans le domaine de la robotique mobile. Une bonne segmentation des salles est la base d'une bonne cartographie topologique/sémantique, d'une bonne catégorisation des endroits ou d'une bonne navigation autonome. Il existe plusieurs algorithmes pour diviser une carte architecturale selon les types suivants :

— La segmentation complètement automatique et la segmentation interactive, ce qui demande l'intervention de l'utilisateur pour pouvoir bien déterminer les salles.
— La segmentation *offline* sur des cartes connues *a priori* et celle *online* qui est faite durant le SLAM
— La segmentation peut se diviser par rapport à l'approche utilisée : par graphe de Voronoi, par morphologique, par la transformation de distances, par apprentissage ou par interprétation de la carte architecturale.

Dans la limite du projet, nous étudions seulement les algorithmes de segmentation complètement automatique car on veut simplifier le plus possible les tâches des utilisateurs qui ne sont pas souvent habitués aux nouvelles technologies. La segmentation considérée est celle

offline car on possède déjà une carte architecturale de l'environnement avant de naviguer. L'approche par l'interprétation de la carte architecturale ne sera pas traitée car on veut pouvoir aussi traiter les cartes de l'environnement faites par les algorithmes de SLAM. Pour la suite, nous allons évaluer différents algorithmes de segmentation de salles.

4.2.1 Approches pour la segmentation de salles

Approche par la transformation de distances

L'idée principale de cette approche est comme suit :

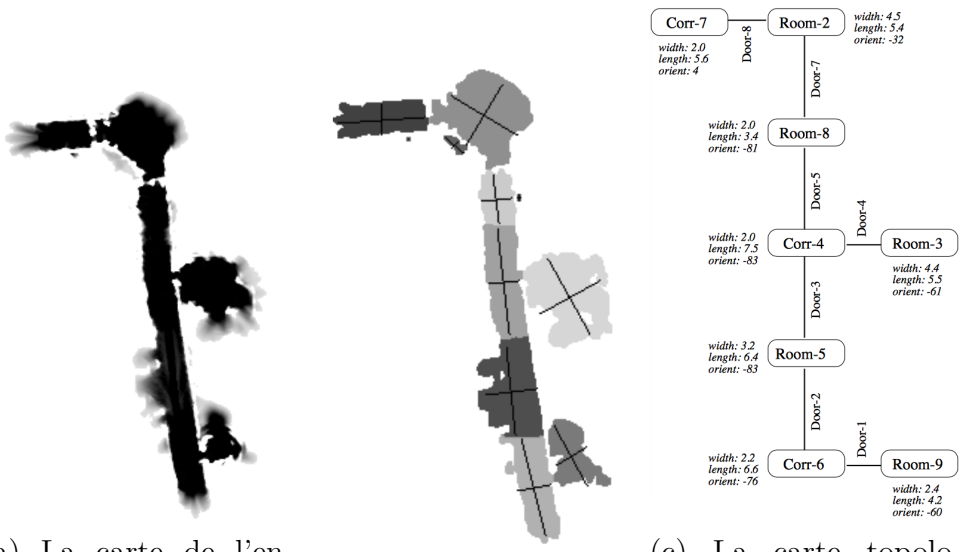
- La transformation de distances est une carte qui représente la distance des pixels blancs (position vide) au pixel noir (position occupée) le plus proche.
- On peut obtenir ensuite des maximums locaux dans cette carte.
- Les pixels qui tendent vers le même maximum local par augmentation du gradient appartiennent à la même salle.

L'approche de segmentation de salles par transformation de distance a été introduite par Diosi et al. (2005). C'est un algorithme semi-autonome qui demande à l'utilisateur de fournir la position des salles en amont. L'algorithme de segmentation des salles sera appliqué *offline* après la cartographie. Spexard et al. (2006) propose ensuite un algorithme interactif pour les robots assistants dans une maison qui est capable de distinguer les différentes pièces dans une maison (cuisine, chambre, salon, etc.). Le premier algorithme complètement autonome a été développé par Topp and Christensen (2010). Il détecte les transitions entre des salles (les portes) pour pouvoir décider de la localisation des salles. Nieto-Granda et al. (2010) proposent une méthode pour distinguer les salles et les couloirs automatiquement. Son algorithme peut alors construire de façon complètement autonome une carte topologique de l'environnement.

Approche morphologique

Fabrizi and Saffiotti (2002) et Buschka and Saffiotti (2002) ont proposé l'algorithme de segmentation par morphologie qui est capable de diviser la carte en plusieurs partitions et les classes en couloirs et salles en utilisant les caractéristiques morphologiques. L'idée principale de cette approche est décrite à la figure 4.2 :

- Utiliser la dilatation et l'érosion pour diviser la carte 4.2a en plusieurs partitions 4.2b.
- Calculer les moments de chaque partition pour en estimer les caractéristiques : le centre, l'orientation, la longueur, la largeur et l'excentricité
- Diviser les partitions en couloirs et salles suivant les caractéristiques précédentes 4.2c.



(a) La carte de l'environnement (les pixels noirs sont libres)

(b) La carte des partitions

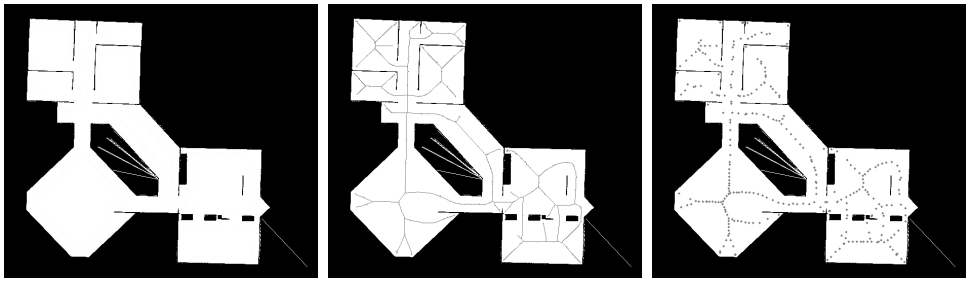
(c) La carte topologique de l'environnement

Figure 4.2 Étapes de la segmentation par l'approche morphologique (Fabrizi and Saffiotti, 2002).

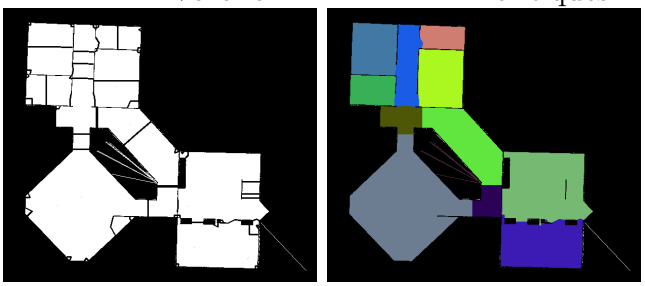
Approche par graphe de Voronoi

C'est l'approche la plus populaire pour faire la segmentation de salles. Elle est basée sur le graphe de Voronoi déjà présenté à la section 4.2.1. Thrun (1998) utilise le graphe de Voronoi la première fois pour partitionner une carte. La méthode est utilisée dans l'article de Wurm et al. (2008) pour résoudre le problème de cartographie avec plusieurs robots. L'optimisation dans ce cas est faite pour choisir les points critiques seulement sur les vraies portes en prenant compte de la structure de l'environnement. Beeson et al. (2005) ont défini le graphe de Voronoi élargi en suivant les murs dans les larges salles, ce qui peut résoudre le problème illustré à la figure 4.6. L'idée principale de l'algorithme de graphe de Voronoi est illustrée à la figure 4.4

— Calculer le graphe de Voronoi pour trouver les points à équidistance de trois obstacles en même temps (Fig. 4.4b).
— Trouver les points critiques dont la distance à l'obstacle est un minimum local (Fig. 4.4c). L'idée est de trouver un passage étroit dans la carte (la porte).
— Diviser la carte en plusieurs petites partitions suivant les lignes critiques trouvées. Les lignes critiques sont créées en connectant les points critiques avec ces deux obstacles le plus proches (Fig. 4.4d).
— Fusionner les partitions pour obtenir les salles dans la carte (Fig. 4.4e).



(a) La carte de l'environnement (b) Le graphe de Voronoi (c) La carte des points critiques



(d) La carte des partitions (e) Le résultat avec des salles

Figure 4.4 Étapes de la segmentation par l'approche par graphe de Voronoi (Bormann et al., 2016).

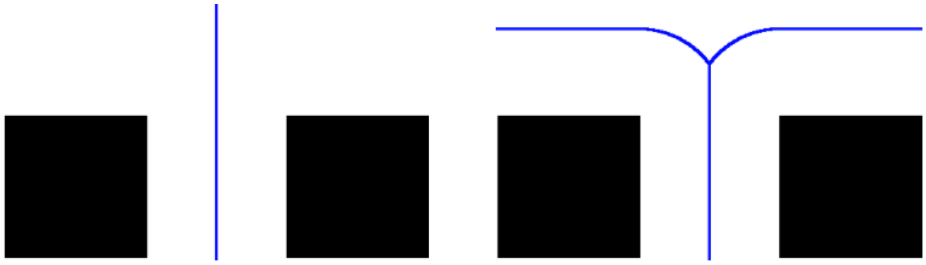


Figure 4.6 L'image à gauche montre le comportement du graphe de Voronoi en rentrant dans une grande salle. Si la portée des capteurs est petite, cette action amène le robot au milieu de la salle où il n'y a pas beaucoup de caractéristiques pour faire la localisation et la cartographie. L'image à droite est le comportement du graphe de Voronoi élargi qui peut résoudre le problème en suivant le mur. (Beeson et al., 2005)

Approche par apprentissage

L'approche par apprentissage utilise souvent une ou des librairies pour s'entraîner et extraire les caractéristiques nécessaires dans la carte et après les utiliser pour trouver et classifier les salles et les couloirs. C'est donc un apprentissage supervisé. Martinez Mozos et al. (2006) utilisent le classificateur *AdaBoost* (Freund and Schapire, 1995) [1] pour analyser les données d'entraînements. Avec les données du LIDAR, *AdaBoost* peut décider des caractéristiques nécessaires pour classifier la carte. Ensuite, en utilisant un *Hidden Markov Model* (HMM), il connecte la sortie de *AdaBoost* à la trajectoire du robot pour distinguer les salles, les couloirs et les portes dans la carte. La performance de l'algorithme de Martinez Mozos et al. (2006) dépend fortement de la trajectoire du robot.

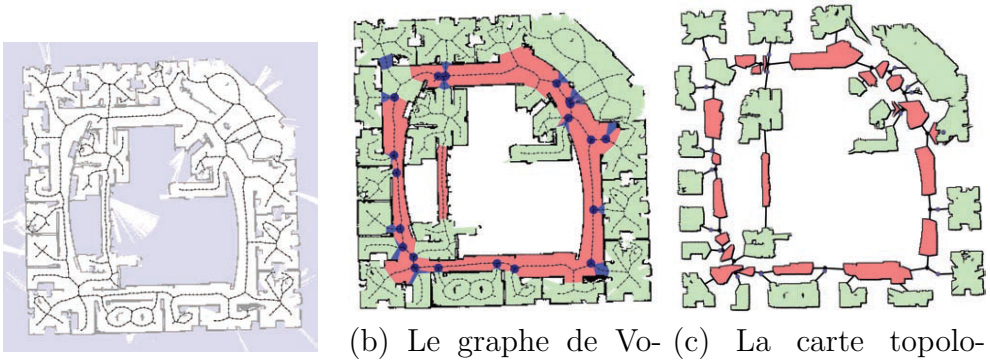
Friedman et al. (2007) ont proposé une amélioration de la méthode de Martinez Mozos et al. (2006) en utilisant le graphe de Voronoi qui peut résoudre ce problème. Il applique le classificateur à la sortie de la segmentation de Voronoi pour obtenir un étiquetage qui ne dépend pas de la trajectoire du robot. La figure 4.8 décrit les 3 étapes de cette combinaison. Le résultat final est moins bruité que celui de Martinez Mozos et al. (2006).

Ekvall et al. (2007) décrit une méthode pour partitionner la carte en détectant les portes et les espaces étroits. Il utilise le descripteur Scale Invariant Feature Transform (SIFT) qui détecte et compare les points les plus invariants (par translation, rotation, mise à l'échelle, etc.) dans l'images.

Approche par interprétation de la carte architecturale

La dernière approche présentée prend à l'entrée une carte architecturale pour diviser les salles en profitant des notations et des symboles précis de la carte. Ahmed et al. (2012), de las Heras et al. (2014), Ahmed et al. (2012) applique cette méthode dans leurs travaux, tandis que Capobianco et al. (2016) utilisent le détecteur *Canny* et les transformations de Hough pour trouver les murs et diviser le plan. Ahmed et al. (2012) et de las Heras et al. (2014) utilisent une détection des lignes droites pour chercher les murs, construire la carte d'occupation et en même temps chercher les portes et classifier les salles. Avec la carte architecturale, on peut ajouter les noms des salles dans la carte topologique créée. La procédure de l'algorithme de Ahmed et al. (2012) est décrite à la figure 4.10. On constate que la précision et la qualité de la carte de cette méthode est remarquable comparativement au autres.

1. *Adaptive Boosting* est un meta-algorithme de l'apprentissage supervisé formulé qui peut combiner plusieurs algorithmes d'apprentissages pour améliorer sa performance.



(a) La carte de l'environnement avec le graphe de Voronoi

(b) Le graphe de Voronoi classifié : les salles (vert). les couloirs (rouge), les transitions (bleu foncé)

(c) La carte topologique de l'environnement

Figure 4.8 Étapes de la segmentation par apprentissage de Friedman et al. (2007).

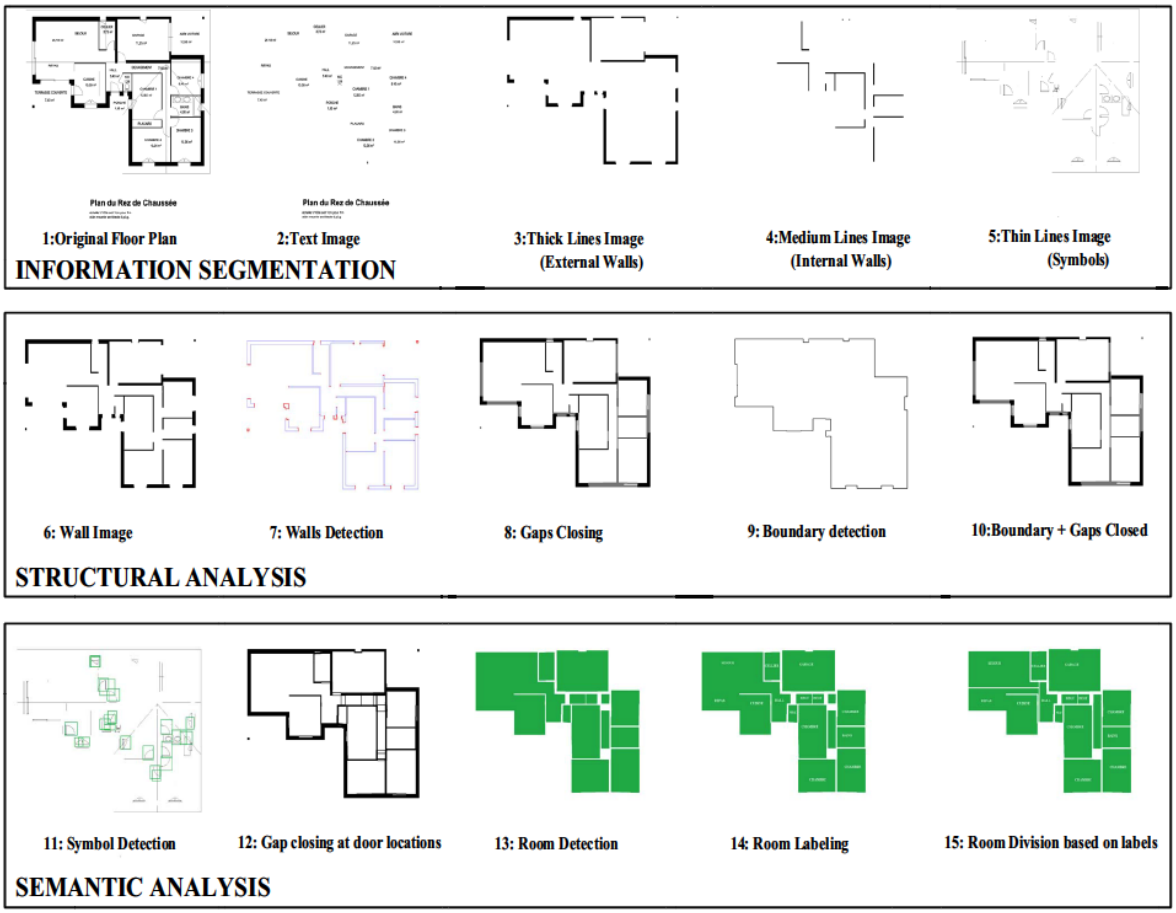


Figure 4.10 Étapes dans l'algorithme de Ahmed et al. (2012)

4.2.2 L'algorithme choisi

Pour choisir l'algorithme le mieux adapté à notre programme, on utilise les résultats de Bormann et al. (2016) qui ont fait une évaluation des algorithmes de segmentation par les approches précédentes. L'approche par interprétation de la carte architecturale n'est pas considérée dans la comparaison car elle utilise une entrée spécifique qui est beaucoup plus précise avec moins d'incertitude que les autres. Dans notre problème, on veut extraire les salles à partir de la carte architecturale mais aussi pour une carte faite par les algorithmes de SLAM. L'utilisation seul de la carte architecturale n'est donc pas considérée. Les résultats d'analyse des différents algorithmes dans la suite sont pris de l'article de Bormann et al. (2016). Les quatre approches présentées ci-dessus sont comparées (sans la dernière méthode). Après avoir choisi le bon algorithme, un test avec notre base de données sera fait pour re-valider ce choix.

Tableau 4.1 Précision et rappel des algorithmes avec les cartes avec et sans meubles

		Morphologique	Distance	Voronoi	Apprentissage
Sans meubles	Rappel	$98.1\% \pm 2.4\%$	$96.9\% \pm 2.8\%$	$95.0\% \pm 2.3\%$	$89.2\% \pm 11.8\%$
	Précision	$88.5\% \pm 9.2\%$	$88.4\% \pm 9.3\%$	$94.8\% \pm 5.0\%$	$90.4\% \pm 8.0\%$
Avec meubles	Rappel	$84.6\% \pm 7.2\%$	$76.1\% \pm 12.3\%$	$86.6\% \pm 5.2\%$	$85.1\% \pm 7.2\%$
	Précision	$90.5\% \pm 8.1\%$	$88.4\% \pm 8.5\%$	$94.5\% \pm 5.1\%$	$87.1\% \pm 14.5\%$

La figure 4.11 nous donne quelques exemples de résultats de ces algorithmes. En combinant avec le tableau 4.1 qui donne la précision et le rappel des algorithmes, on peut voir la performance des quatre méthodes qualitativement et en même temps quantitativement. La précision est définie comme la surface commune entre la salle segmentée et la salle exacte divisée par la surface segmentée. Le rappel est défini de façon similaire comme le nombre de pixels communs entre les deux salles divisé par le nombre de pixels de la salle dans la carte exacte. Une carte est bien segmentée si les deux indices sont hauts. Avec les cartes sans meubles, la segmentation morphologique obtient le meilleur rappel, mais une plus petite précision que la méthode de Voronoi. Dans le cas avec des meubles, la meilleure méthode est celle de Voronoi avec les meilleurs précision et rappel, et la plus petite déviation. On peut faire la même observation en analysant la figure 4.11. La méthode d'apprentissage peut bien traiter les cartes plus complexes avec meubles mais avoir une mauvaise précision dans plusieurs cas. Les méthodes morphologique et transformation de distances détectent souvent les salles plus grandes qu'en réalité et même fusionnent plusieurs salles ensemble. La méthode de Voronoi donne une bonne qualité globale. Le seul défaut est qu'elle divise souvent les couloirs en plusieurs petits segments.

Le tableau 4.2 nous montre le temps d'exécution des quatre algorithmes avec différents

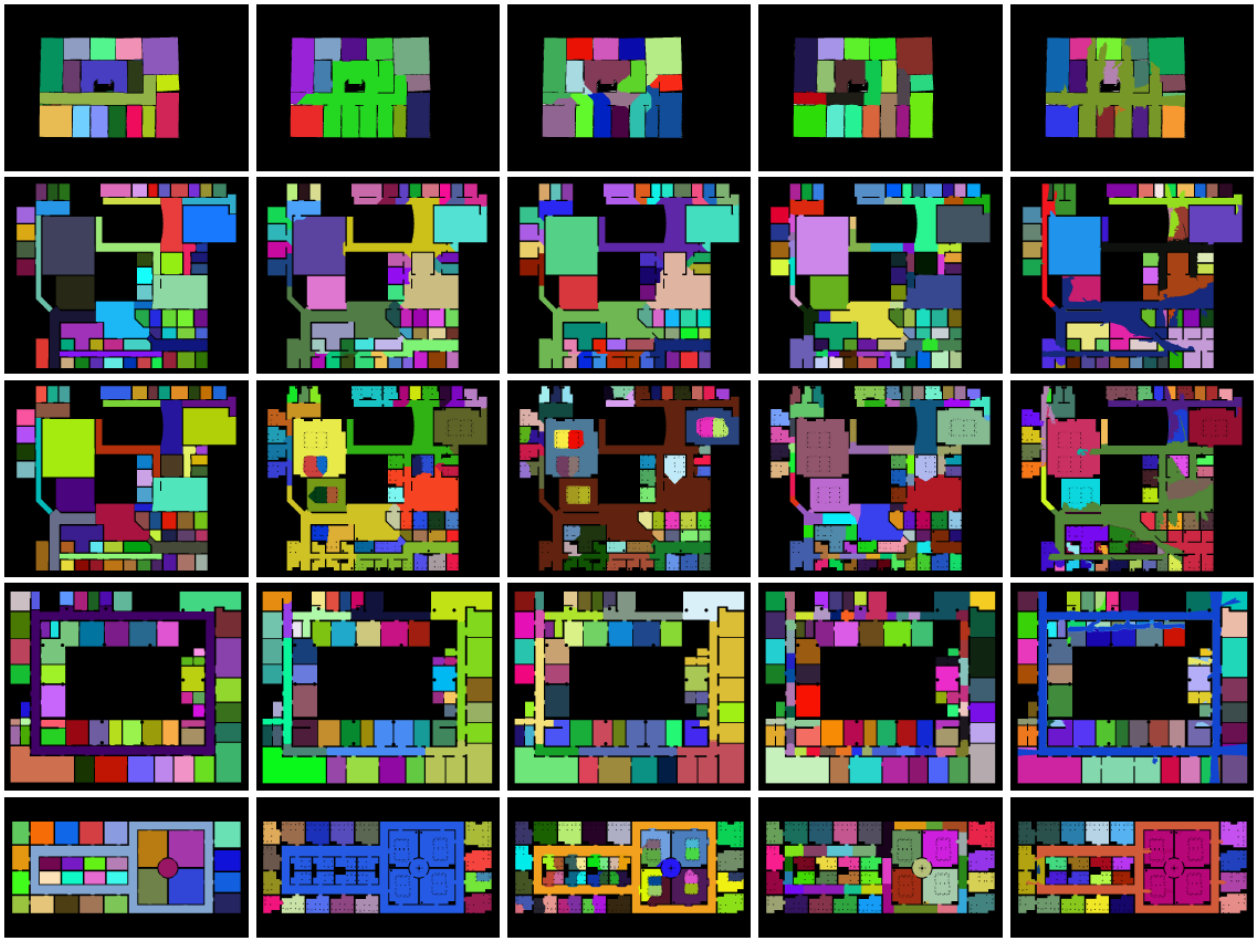


Figure 4.11 Exemples des quatre algorithmes. La première colonne représente les cartes exactes de l'environnement, la deuxième, les résultats de l'algorithme par morphologie, la troisième, ceux de la méthode par transformation de distance, la quatrième, ceux du graphe de Voronoi, et la dernière, ceux de l'algorithme par apprentissage. Les cartes exactes sont étiquetées manuellement afin de séparer les salles et les couloirs.

Tableau 4.2 Temps d'exécution moyen des algorithmes avec les cartes avec et sans meubles

	Morphologique	Transformation de distances	Graphe de Voronoi	Apprentissage
Sans meubles	$1.6 \pm 2.6\,\mathrm{s}$	$1.8 \pm 2.7\,\mathrm{s}$	$13.0 \pm 15.3\,\mathrm{s}$	$269.3 \pm 196.7\,\mathrm{s}$
Avec meubles	$1.1 \pm 1.2\,\mathrm{s}$	$1.3 \pm 1.4\,\mathrm{s}$	$12.0 \pm 14.2\,\mathrm{s}$	$245.0 \pm 171.1\,\mathrm{s}$

types de cartes. Tous les algorithmes prennent un peu plus de temps pour traiter les cartes avec meubles car la complexité de la carte augmente avec les meubles. On voit que le temps de traitement des deux méthodes par morphologie et transformation de distances sont le plus bas, aux alentours de $1-2\,\mathrm{s}$. Celui du graphe de Voronoi est d'environ $13\,\mathrm{s}$. Enfin l'algorithme basé sur l'apprentissage utilise plus de quatre minutes pour traiter chaque carte.

En conclusion, la méthode basée sur le graphe de Voronoi donne les résultats de bonne qualité globale avec un temps d'exécution raisonnable. Dans certain cas, la méthode par apprentissage peut donner une réponse plus précise. Mais le temps d'exécution est trop élevé et ne répond pas à nos besoins (on ne veut pas attendre cinq minutes avant de bouger en rentrant dans une nouvelle pièce). Le défaut de la méthode Voronoi, soit la division du couloir en plusieurs fragments, est tolérable car on veut aussi avoir des points intermédiaires dans la trajectoire.

4.3 Nouvelle structure du système

4.3.1 Structure de la commande collaborative

La nouvelle structure de la commande collaborative est présentée à la figure 4.12. Deux nouveaux modules sont introduits dans le programme. Le premier module est la segmentation

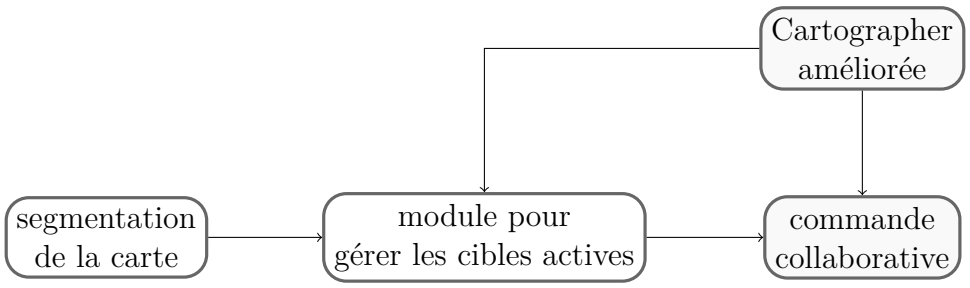


Figure 4.12 Structure de la commande collaborative

de salles qui détecte tous les points intéressants sur la carte, qui peuvent ainsi devenir une destination ou un point de passage du robot. Deuxièmement, on introduit le module pour gérer les cibles actives sur la carte. Il prend en compte la position actuelle du robot et fournit les cibles les plus proches qui sont accessibles. Une fois ces cibles établies, il fournit ces destinations au module de commande collaborative.

4.3.2 Module de segmentation de salles

La méthode de segmentation suit bien les étapes présentées dans la section 4.2.1. Dans la transition de l'étape 3 (Figs 4.4d et 4.14c) à l'étape 4 (Figs 4.4e, 4.14g et 4.14h), nous devons réduire le nombre des lignes critiques afin de fusionner les partitions dans la carte. Les conditions pour combiner les petits segments de la carte sont :

— Assigner des étiquettes à chaque région dans la carte dont la surface n'est pas trop petite ($< 2m^2$) ni trop grande ($> 120m^2$). Les régions qui ne sont pas étiquetées sont fusionnées avec leur voisin. Le résultat de cette étape est illustré à la figure 4.14d.

— Les petites surfaces ($< 12.5m^2$) avec un seul voisin et leurs frontières possèdant moins de 75% du mur sont fusionnées avec leurs voisins comme illustrés à la figure 4.14e. Le fait qu'il n'y ait pas assez de murs au bord de la surface signifie qu'elle n'est pas une région fermée.

— Chercher la zone avec un seul voisin qui possède au maximum deux voisins. Si la grande zone a plus que 50% de périmètres qui est le mur, ce qui est souvent le cas de deux partitions dans une seule salle, on peut fusionner les deux zones. Cette étape est illustrée à la figure 4.14f.

— Combiner les régions qui partagent beaucoup leur périmètre ($> 20\%$ pour les petites salles et $> 10\%$ pour les grandes salles). Cette étape peut résoudre des problèmes de fragmentation des couloirs et salles (Fig. 4.14g).

— Fusionner les régions dont plus de 40% de leurs bords sont partagés avec l'autre. C'est souvent le cas autour des meubles ouvertes.

Nous testons l'algorithme avec l'environnement de test de la figure 3.15. La procédure est illustré à la figure 4.14. Le temps de traitement d'une telle carte ($20 \times 20\,\mathrm{m}^2$) est de 8.2 s, ce qui est rapide. Mais notons que nous devons aussi travailler avec des environnements beaucoup plus grands (par exemple, le pavillon Lassonde de Polytechnique Montréal a une carte architecturale de dimension environ $120 \times 87\,\mathrm{m}^2$ ce qui est traité en 161.5 s soit approximativement 3 min). Le temps de traitement de l'algorithme est à la limite dans ce cas là. La figure 4.14h nous montre le résultat à la sortie du module. On peut voir que toutes les salles dans la carte sont bien détectées et séparées.

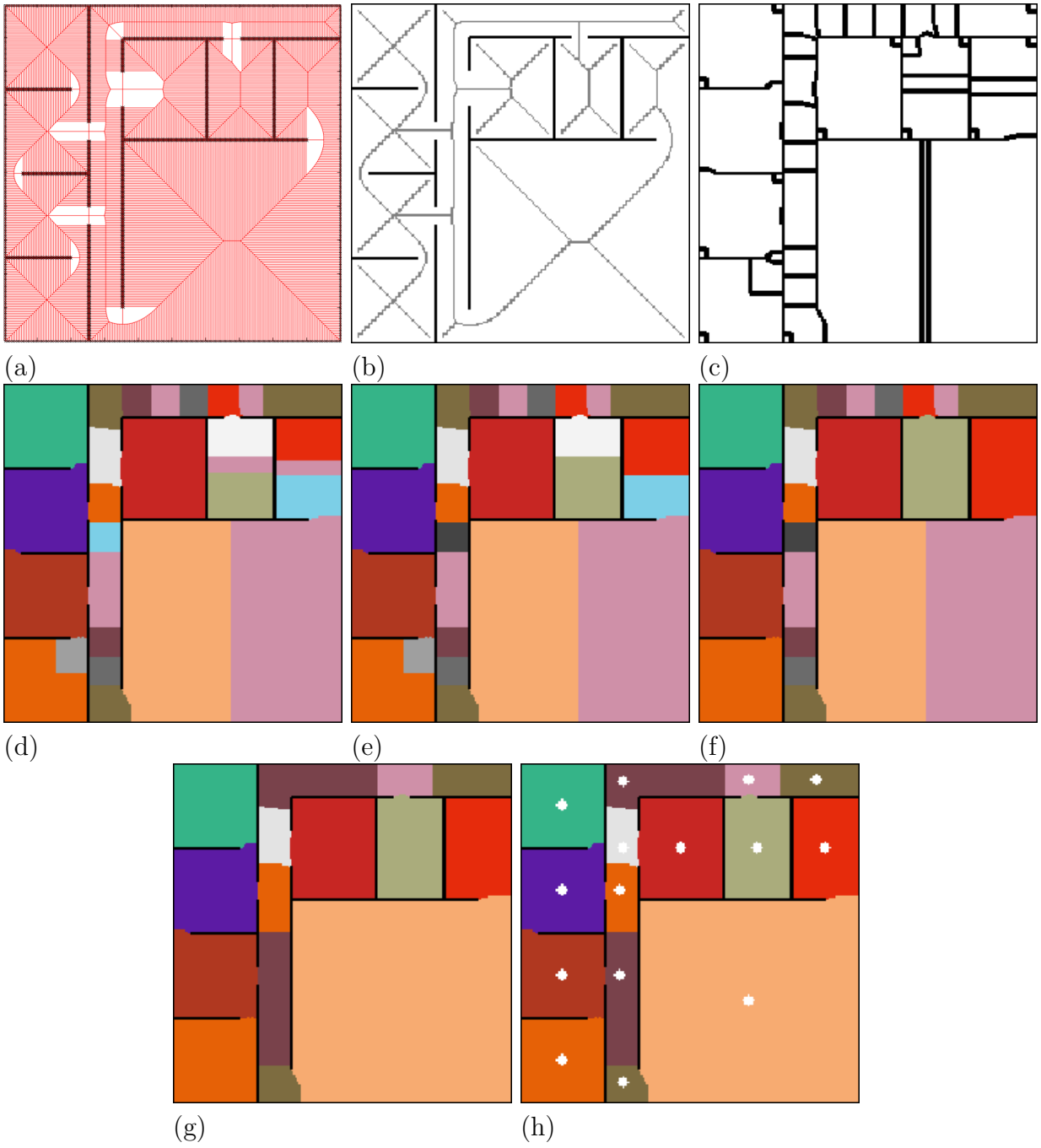


Figure 4.14 Procédure de segmentation de la carte de l'environnement de test

4.3.3 Module pour gérer les cibles actives

Après avoir déterminé les destinations dans la carte, on veut maintenant les insérer dans l'algorithme de commande partagée. Par contre, si on ajoute plusieurs destinations en même temps, comme dans la figure 4.16, on ajoute aussi des destinations de très basses probabilités. Cela peut alourdir la charge de calcul dans le système ; nous devons alors concevoir une solution pour ne pas avoir trop de destinations dans l'environnement simultanément.

La fonction principale de ce module est de limiter les destinations sur la carte. Dès l'obtention de la carte segmentée, nous allons l'analyser et créer un graphe topologique de l'environnement. Chaque nœud dans le graphe est un point sur la carte, et il sauvegarde le nombre et la position de ses voisins. En utilisant la transformation entre l'origine de la carte créée par le SLAM et la carte de l'environnement dans la section précédente 3.2, on peut faire une transformation pour placer les destinations à la bonne position dans la carte et on peut aussi déterminer dans quelle salle est le robot. Quand le robot approche d'un nœud, ce point sera désactivé et les voisins seront activés sur la carte. Avec cette approche, le nombre maximal de destinations sur la carte est le nombre maximal de voisins d'un nœud.

4.4 Résultat du programme

Pour tester notre programme, nous utilisons le mouvement du robot dans l'environnement de test à la figure 3.15 avec la commande collaborative. Le mouvement du robot est du point A au point B comme à la figure 4.15.

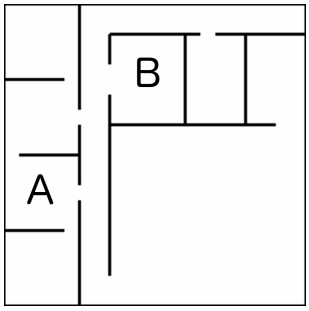


Figure 4.15 Trajectoire de test du robot

La figure 4.18a montre les états d'exécution de notre programme de tests en simulation. La première étape est de déterminer la position du robot dans la carte connue de l'environnement avec l'algorithme AMCL. Après avoir estimé la position actuelle du robot, la carte est transférée à *Google Cartographer* pour faire le SLAM. L'algorithme proposé dans la section 4.3.3 va ensuite gérer les destinations pour amener l'utilisateur vers la destination souhaitée tel qu'illustré aux figures 4.18b, 4.18c, 4.18d, 4.18e, 4.18f. On peut voir que à chaque état, le nombre de cibles est limité, ce qui permet de diminuer la puissance de calcul requise lors de la mise en œuvre. La figure 4.18e montre que le robot arrive avec succès à la destination finale.

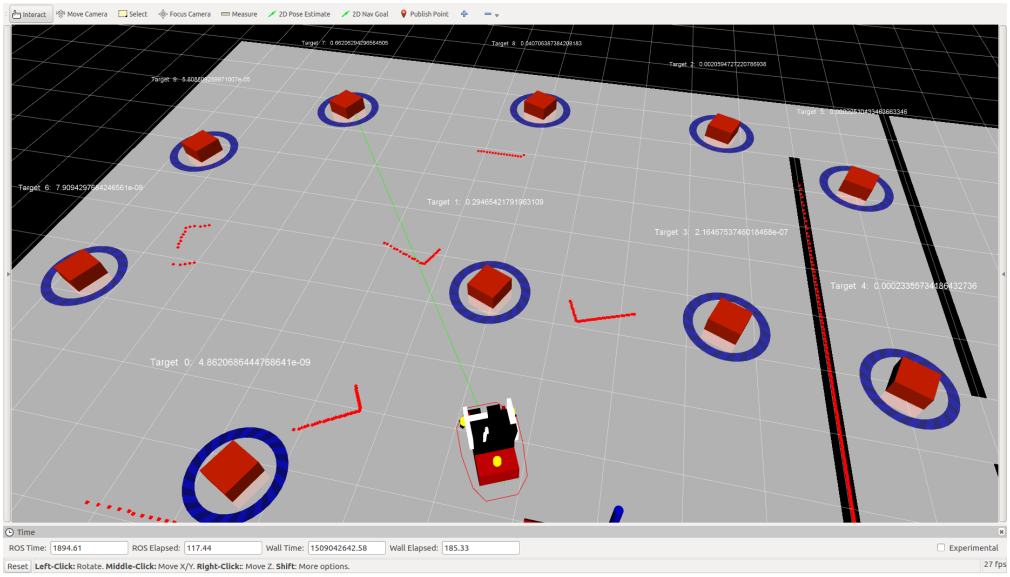


Figure 4.16 Placement de plusieurs destinations dans la carte

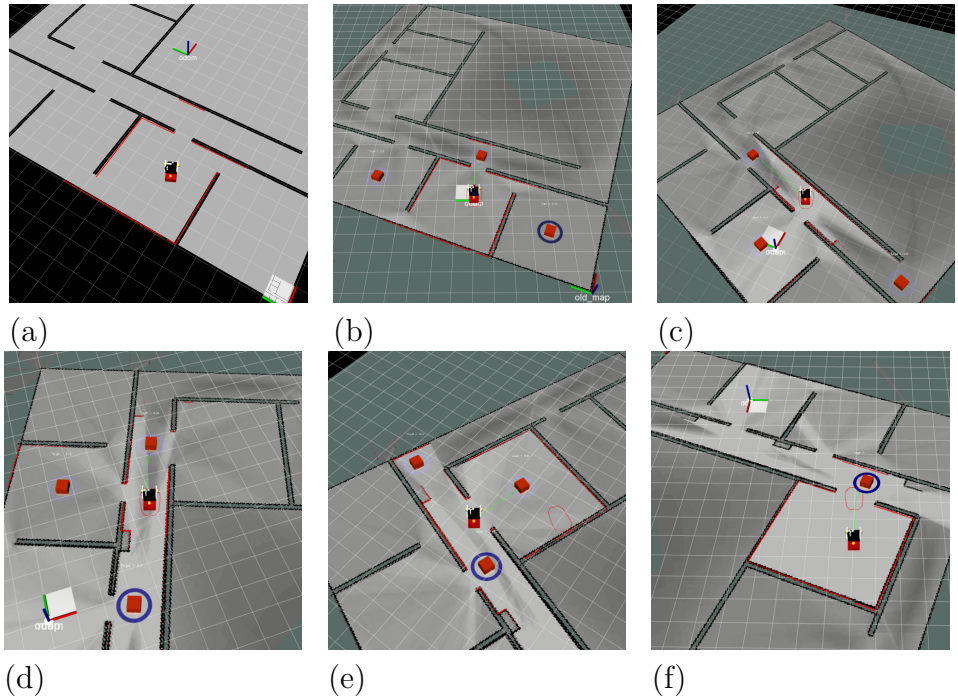


Figure 4.18 États de notre programme final

CHAPITRE 5 CONCLUSION

Le projet a permis de compléter la commande collaborative sur un fauteuil roulant électrique intelligent. Il s'agit d'une amélioration d'un module de cartographie et localisation simultanée et d'une implémentation d'un module de segmentation de cartes pour notre FRMI. Ce travail contribue au domaine de l'interaction entre l'homme et le robot par des solutions testées d'après les résultats présentés dans les chapitres précédents. Ce projet a pour objectif principal d'apporter l'autonomie et le confort à la vie des personnes âgées et à mobilité réduite afin d'augmenter leur qualité de vie.

5.1 Synthèse des travaux

Les principales réalisations de ce projet sont :

Amélioration de l'algorithme de SLAM

- La comparaison des trois algorithmes de SLAM 2D dans ROS et le choix d'une technique de SLAM à utiliser sur notre chaise.
- L'inclusion de la carte *a priori* dans le SLAM par la conception d'un module de génération des données virtuelles d'odométries et de LIDAR pour créer une trajectoire virtuelle. Le module doit fournir une information complète de l'environnement sans surcharger l'algorithme avec trop de données dans la trajectoire. Après le chargement des données du SLAM virtuel, *Cartographer* permet de compléter la cartographie en ajoutant des obstacles absents dans le plan *a priori*.
- Le module permet à la commande collaborative de prendre en compte des destinations dans les régions qui ne sont pas encore explorées et calculer la trajectoire globale vers ces destinations sans rentrer dans les impasses.
- Le module de génération de trajectoire est indépendant du module de *Cartographer* et il est alors utilisable avec des autres algorithmes de SLAM basé sur la représentation par un graphe.

Segmentation des salles

- La comparaison des techniques de segmentation de cartes 2D et le choix d'un algorithme afin d'implémenter sur notre chaise. Les salles et les couloirs segmentées seront pris pour créer un ensemble de destinations pour l'algorithme de commande collaborative.
- La conception d'un module qui nous permet de mettre les destinations dans la carte et de désactiver les destinations qui ne sont pas raisonnables.
- Le module permet à la commande collaborative de déterminer et gérer les destinations automatiquement sans avoir besoin d'intervention de l'utilisateur.
- Le module de segmentation des cartes est indépendant de tous les autres modules du FRMI. Il peut donc être exécuté en parallèle pour traiter d'autres cartes (par exemple d'autres étages dans l'environnement) pendant que l'utilisateur continue sa navigation.

5.2 Limitations

Les solutions proposées améliorent l'interaction entre l'utilisateur et la chaise roulante en simplifiant les tâches du pilote. Les résultats des tests en simulation sont prometteurs. Par contre, ils existent quelques limites qui demandent des améliorations futures.

Limitation du SLAM dans un environnement partiellement connu Le SLAM dans un environnement partiellement connu dépend du respect de la carte architecturale pendant la construction. L'amélioration de la carte dépend de la qualité des capteurs du robot. Donc, si nous prenons un plan médiocre, on ne peut pas obtenir une bonne session de cartographie à la fin. D'autre part, avec un défaut d'odométrie qui fait dévier la chaise, il n'est pas possible de fermer la boucle et de faire le *loop-closure*.

Limitation de la segmentation de carte Le centre de la salle segmentée n'est pas forcément l'endroit où l'utilisateur veut arriver dans la salle. Donc avec les grandes salles avec plusieurs obstacles, il faudrait encore diviser en plusieurs partitions pour avoir des meilleurs résultats. La position des salles dans la carte dépend fortement du SLAM. S'il y a une déviation, les positions des destinations dans la carte seront aussi déplacées.

5.3 Améliorations futures

Améliorations pour la cartographie

- Améliorer le SLAM en augmentant la covariance de la carte *a priori*. La position de la carte *a priori* est fixe et ne sera plus modifiée.
- Prendre en compte seulement des obstacles dans la carte architecturale et ignorer les régions libres (car les murs dans les cartes architecturales sont fixes).
- Corriger l'odométrie du fauteuil pour mieux apparier la carte *a priori* et la carte créée par le SLAM.

Améliorations pour la segmentation de la carte

- Si on rentre dans une grande salle, on peut faire des segmentations de cette salle avec la carte mise à jour pour trouver les obstacles et des destinations potentielles dans la salle.
- Faire des segmentations en arrière-plan pour traiter les autres cartes des régions autour du robot.
- Quand on fait une fermeture de boucle et que la position des salles changent, on devrait procéder à une correction pour remettre les destinations aux bons endroits.

RÉFÉRENCES

P. Agarwal, G. D. Tipaldi, L. Spinello, C. Stachniss, et W. Burgard, "Robust map optimization using dynamic covariance scaling", dans *Robotics and Automation (ICRA), 2013 IEEE International Conference on*. IEEE, 2013, pp. 62–69.

S. Agarwal, K. Mierle, et Others, "Ceres solver", `http://ceres-solver.org`.

S. Ahmed, M. Liwicki, M. Weber, et A. Dengel, "Automatic room detection and room labeling from architectural floor plans", dans *Document Analysis Systems (DAS), 2012 10th IAPR International Workshop on*. IEEE, 2012, pp. 339–343.

L. A. Andersson et J. Nygards, "C-sam : Multi-robot slam using square root information smoothing", dans *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*. IEEE, 2008, pp. 2798–2805.

C. Andrieu, A. Doucet, et E. Punskaya, "Sequential monte carlo methods for optimal filtering", dans *Sequential Monte Carlo Methods in Practice*. Springer, 2001, pp. 79–95.

Autodesk, "Dxf reference", `https://www.autodesk.com/techpubs/autocad/acadr14/dxf/dxf_reference.htm`.

P. Beeson, N. K. Jong, et B. Kuipers, "Towards autonomous topological place detection using the extended voronoi graph", dans *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*. IEEE, 2005, pp. 4373–4379.

P. J. Besl, N. D. McKay *et al.*, "A method for registration of 3-d shapes", *IEEE Transactions on pattern analysis and machine intelligence*, vol. 14, no. 2, pp. 239–256, 1992.

R. Bormann, F. Jordan, W. Li, J. Hampp, et M. Hägele, "Room segmentation : Survey, implementation, and analysis", dans *Robotics and Automation (ICRA), 2016 IEEE International Conference on*. IEEE, 2016, pp. 1019–1026.

P. Buschka et A. Saffiotti, "A virtual sensor for room detection", dans *Intelligent Robots and Systems, 2002. IEEE/RSJ International Conference on*, vol. 1. IEEE, 2002, pp. 637–642.

R. Capobianco, G. Gemignani, D. D. Bloisi, D. Nardi, et L. Iocchi, "Automatic extraction of structural representations of environments", dans *Intelligent Autonomous Systems 13*. Springer, 2016, pp. 721–733.

L. Carlone, M. K. Ng, J. Du, B. Bona, et M. Indri, “Rao-blackwellized particle filters multi robot slam with unknown initial correspondences and limited communication”, dans *Robotics and Automation (ICRA), 2010 IEEE International Conference on.* IEEE, 2010, pp. 243–249.

L. Carlone, R. Aragues, J. A. Castellanos, et B. Bona, “A linear approximation for graph-based simultaneous localization and mapping”, *Robotics : Science and Systems VII*, pp. 41–48, 2012.

J. A. Castellanos, J. Neira, et J. D. Tardós, “Limits to the consistency of ekf-based slam”, *IFAC Proceedings Volumes*, vol. 37, no. 8, pp. 716–721, 2004.

Y. Chen et G. Medioni, “Object modeling by registration of multiple range images”, dans *Robotics and Automation, 1991. Proceedings., 1991 IEEE International Conference on.* IEEE, 1991, pp. 2724–2729.

T. Davis, “Direct methods for sparse linear systems (fundamentals of algorithms 2) society for industrial and applied mathematics”, 2006.

L.-P. de las Heras, S. Ahmed, M. Liwicki, E. Valveny, et G. Sánchez, “Statistical segmentation and structural recognition for floor plan interpretation”, *International Journal on Document Analysis and Recognition (IJDAR)*, vol. 17, no. 3, pp. 221–237, 2014.

F. Dellaert et M. Kaess, “Square root sam : Simultaneous localization and mapping via square root information smoothing”, *The International Journal of Robotics Research*, vol. 25, no. 12, pp. 1181–1203, 2006.

A. Diosi, G. Taylor, et L. Kleeman, “Interactive slam using laser and advanced sonar”, dans *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on.* IEEE, 2005, pp. 1103–1108.

A. Doucet, “On sequential simulation-based methods for bayesian filtering”, 1998.

A. Doucet, N. De Freitas, K. Murphy, et S. Russell, “Rao-blackwellised particle filtering for dynamic bayesian networks”, dans *Proceedings of the Sixteenth conference on Uncertainty in artificial intelligence.* Morgan Kaufmann Publishers Inc., 2000, pp. 176–183.

S. Ekvall, D. Kragic, et P. Jensfelt, “Object detection and mapping for service robot tasks”, *Robotica*, vol. 25, no. 2, pp. 175–187, 2007.

E. Fabrizi et A. Saffiotti, “Augmenting topology-based maps with geometric information”, *Robotics and Autonomous Systems*, vol. 40, no. 2, pp. 91–97, 2002.

A. Franchi, G. Oriolo, et P. Stegagno, “Mutual localization in multi-robot systems using anonymous relative measurements”, *The International Journal of Robotics Research*, vol. 32, no. 11, pp. 1302–1322, 2013.

U. Frese, “A discussion of simultaneous localization and mapping”, *Autonomous Robots*, vol. 20, no. 1, pp. 25–42, 2006.

Y. Freund et R. E. Schapire, “A desicion-theoretic generalization of on-line learning and an application to boosting”, dans *European conference on computational learning theory*. Springer, 1995, pp. 23–37.

S. Friedman, H. Pasula, et D. Fox, “Voronoi random fields : Extracting topological structure of indoor environments via place labeling.” dans *IJCAI*, vol. 7, 2007, pp. 2109–2114.

M. Ghorbel, “Commande collaborative pour un fauteuil roulant intelligent”, Mémoire de maîtrise, École Polytechnique de Montréal, 2016.

G. Grisetti, C. Stachniss, et W. Burgard, “Improved techniques for grid mapping with rao-blackwellized particle filters”, *IEEE transactions on Robotics*, vol. 23, no. 1, pp. 34–46, 2007.

G. Grisettiyz, C. Stachniss, et W. Burgard, “Improving grid-based slam with rao-blackwellized particle filters by adaptive proposals and selective resampling”, dans *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*. IEEE, 2005, pp. 2432–2437.

W. Hess, D. Kohler, H. Rapp, et D. Andor, “Real-time loop closure in 2d lidar slam”, dans *Robotics and Automation (ICRA), 2016 IEEE International Conference on*. IEEE, 2016, pp. 1271–1278.

A. Howard, “Multi-robot mapping using manifold representations”, dans *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, vol. 4. IEEE, 2004, pp. 4198–4203.

S. Huang et G. Dissanayake, “Convergence and consistency analysis for extended kalman filter based slam”, *IEEE Transactions on robotics*, vol. 23, no. 5, pp. 1036–1049, 2007.

V. Indelman, E. Nelson, N. Michael, et F. Dellaert, “Multi-robot pose graph localization and data association from unknown initial relative poses via expectation maximization”, dans *Robotics and Automation (ICRA), 2014 IEEE International Conference on.* IEEE, 2014, pp. 593–600.

D. S. Johnson et L. A. McGeoch, “The traveling salesman problem : A case study in local optimization”, *Local search in combinatorial optimization*, vol. 1, pp. 215–310, 1997.

M. Kaess, A. Ranganathan, et F. Dellaert, “isam : Fast incremental smoothing and mapping with efficient data association”, dans *Robotics and Automation, 2007 IEEE International Conference on.* IEEE, 2007, pp. 1670–1677.

K. Konolige et K. Chou, “Markov localization using correlation”, dans *IJCAI*, vol. 99, 1999, pp. 1154–1159.

K. Konolige, G. Grisetti, R. Kümmerle, W. Burgard, B. Limketkai, et R. Vincent, “Efficient sparse pose adjustment for 2d mapping”, dans *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on.* IEEE, 2010, pp. 22–29.

A. H. Land et A. G. Doig, “An automatic method of solving discrete programming problems”, *Econometrica : Journal of the Econometric Society*, pp. 497–520, 1960.

K. Levenberg, “A method for the solution of certain non-linear problems in least squares”, *Quarterly of applied mathematics*, vol. 2, no. 2, pp. 164–168, 1944.

J. S. Liu et R. Chen, “Sequential monte carlo methods for dynamic systems”, *Journal of the American statistical association*, vol. 93, no. 443, pp. 1032–1044, 1998.

F. Lu et E. Milios, “Globally consistent range scan alignment for environment mapping”, *Autonomous robots*, vol. 4, no. 4, pp. 333–349, 1997.

G. Majdoub, “Techniques d’optimisation de la navigation globale d’un fauteuil roulant motorisé intelligent”, Mémoire de maîtrise, École Polytechnique de Montréal, 2014.

D. W. Marquardt, “An algorithm for least-squares estimation of nonlinear parameters”, *Journal of the society for Industrial and Applied Mathematics*, vol. 11, no. 2, pp. 431–441, 1963.

O. Martinez Mozos, A. Rottmann, R. Triebel, P. Jensfelt, W. Burgard *et al.*, “Semantic labeling of places using information extracted from laser and vision sensor data”, 2006.

M. Montemerlo, S. Thrun, D. Koller, B. Wegbreit *et al.*, “Fastslam : A factored solution to the simultaneous localization and mapping problem”, dans *Aaai/iaai*, 2002, pp. 593–598.

H. P. Moravec, “Sensor fusion in certainty grids for mobile robots”, *AI magazine*, vol. 9, no. 2, p. 61, 1988.

K. P. Murphy, “Bayesian map learning in dynamic environments”, dans *Advances in Neural Information Processing Systems*, 2000, pp. 1015–1021.

C. Nieto-Granda, J. G. Rogers, A. J. Trevor, et H. I. Christensen, “Semantic map partitioning in indoor environments using regional analysis”, dans *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on.* IEEE, 2010, pp. 1451–1456.

E. Olson, “M3rsm : Many-to-many multi-resolution scan matching”, dans *Robotics and Automation (ICRA), 2015 IEEE International Conference on.* IEEE, 2015, pp. 5815–5821.

E. B. Olson, “Real-time correlative scan matching”, dans *Robotics and Automation, 2009. ICRA’09. IEEE International Conference on.* IEEE, 2009, pp. 4387–4393.

S. Saeedi, M. Trentini, et H. Li, “A hybrid approach for multiple-robot slam with particle filtering”, dans *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on.* IEEE, 2015, pp. 3421–3426.

SAEInternational, “Automated driving levels of driving automation are defined in new sae international standard j3016”, Août 2017. En ligne : `http://www.sae.org/misc/pdfs/automated_driving.pdf`

T. Spexard, S. Li, B. Wrede, J. Fritsch, G. Sagerer, O. Booij, Z. Zivkovic, B. Terwijn, et B. Krose, “Biron, where are you ? enabling a robot to learn new places in a real home environment by integrating spoken dialog and visual localization”, dans *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on.* IEEE, 2006, pp. 934–940.

B. Steux et O. El Hamzaoui, “tinyslam : A slam algorithm in less than 200 lines c-language program”, dans *Control Automation Robotics & Vision (ICARCV), 2010 11th International Conference on.* IEEE, 2010, pp. 1975–1979.

R. Tarjan, “Depth-first search and linear graph algorithms”, *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.

S. Thrun, “Learning metric-topological maps for indoor mobile robot navigation”, *Artificial Intelligence*, vol. 99, no. 1, pp. 21–71, 1998.

S. Thrun et M. Montemerlo, “The graph slam algorithm with applications to large-scale mapping of urban structures”, *The International Journal of Robotics Research*, vol. 25, no. 5-6, pp. 403–429, 2006.

S. Thrun, D. Fox, W. Burgard, et F. Dellaert, “Robust monte carlo localization for mobile robots”, *Artificial intelligence*, vol. 128, no. 1-2, pp. 99–141, 2001.

S. Thrun, W. Burgard, et D. Fox, *Probabilistic robotics*. MIT press, 2005.

E. A. Topp et H. I. Christensen, “Detecting region transitions for human-augmented mapping”, *IEEE Transactions on Robotics*, vol. 26, no. 4, pp. 715–720, 2010.

R. Van Der Merwe, A. Doucet, N. De Freitas, et E. A. Wan, “The unscented particle filter”, dans *Advances in neural information processing systems*, 2001, pp. 584–590.

K. M. Wurm, C. Stachniss, et W. Burgard, “Coordinated multi-robot exploration using a segmentation of the environment”, dans *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*. IEEE, 2008, pp. 1160–1165.

X. S. Zhou et S. I. Roumeliotis, “Multi-robot slam with unknown initial correspondence : The robot rendezvous case”, dans *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*. IEEE, 2006, pp. 1785–1792.

ANNEXE A ALGORITHME DE GMAPPING

Require:

S_{t-1}, les particules précédentes

z_t, l'observation récente du LIDAR

u_{t-1}, la mesure d'odométrie récente

Ensure:

S_t, les nouvelles particules

$S_t = \{\}$
for all $s_{t-1}^{(i)} \in S_{t-1}$ **do**
$< x_{t-1}^{(i)}, w_{t-1}^{(i)}, m_{t-1}^{(i)} >= s_{t-1}^{(i)}$
$x_t^{'(i)} = x_{t-1}^{(i)} \bigoplus u_{t-1}$ ▷ La comparaison (*Scan-matching*)
$\hat{x}_t^{(i)} = \arg\max_x p(x|m_{t-1}^{(i)}, z_t, x_t^{'(i)})$
if $\hat{x}_t^{(i)}$ echoue **then**
$x_t^{(i)} \sim p(x_t|x_{t-1}^{(i)}, u_{t-1})$
$w_t^{(i)} = w_{t-1}^{(i)} p(z_t|m_{t-1}^{(i)}, x_t^{(i)})$
else
for $k = 1, ..., K$ **do** ▷ Échantillonner autour $\hat{x}_t^{(i)}$
$x_k \sim \{x_j| \quad |x_j - \hat{x}_t^{(i)}| < \Delta\}$
end for
$\mu_t^{(i)} = (0, 0, 0)^T$ ▷ La distribution Gaussienne
$\eta^{(i)} = 0$
for all $x_j \in \{x_1, ..., x_K\}$ **do**
$\mu_t^{(i)} = \mu_t^{(i)} + x_j p(z_t|m_{t-1}^{(i)}, x_j) p(x_j|x_{t-1}^{(i)}, u_{t-1})$
$\eta^{(i)} = \eta^{(i)} + p(z_t|m_{t-1}^{(i)}, x_j) p(x_j|x_{t-1}^{(i)}, u_{t-1})$
end for
$\mu_t^{(i)} = \mu_t^{(i)} / \eta^{(i)}$
$\Sigma_t^{(i)} = 0$
for all $x_j \in \{x_1, ..., x_K\}$ **do**
$\Sigma_t^{(i)} = \Sigma_t^{(i)} + p(z_t|m_{t-1}^{(i)}, x_j) p(x_j|x_{t-1}^{(i)}, u_{t-1})(x_j - \mu_t^{(i)})(x_j - \mu_t^{(i)})^T$
end for
$\Sigma_t^{(i)} = \Sigma_t^{(i)} / \eta^{(i)}$
$x_t^{(i)} \sim N(\mu_t^{(i)}, \Sigma_t^{(i)})$ ▷ Échantillonnage
$w_t^{(i)} = w_{t-1}^{(i)} \eta^{(i)}$

end if

$m_t^{(i)} = Scanner(m_{t-1}^{(i)}, x_t^{(i)}, z_t)$ ▷ Mise à jour la carte

$S_t = S_t \cup \{< x_t^{(i)}, w_t^{(i)}, m_t^{(i)} >\}$ ▷ Mise à jour des particules

end for

$N_{eff} = \frac{1}{\sum_{i=1}^{N} (w^{(i)})^2}$

if $N_{eff} < N/2$ **then**

$S_t = Ré - échantillonner(S_t)$

end if

ANNEXE B CRÉATION DE LA MATRICE CREUSE $\boldsymbol{H}$

Require:

$\boldsymbol{c_f}$, la liste des nœuds vides

$\boldsymbol{e}$, la liste des contraintes e_{ij}

Ensure:

$\boldsymbol{H}$, la matrice triangulaire supérieure éparse en format CSS

Initialiser le vecteur de taille $\|\boldsymbol{c_f}\|$ du type C++ $std :: map$. Chaque carte représente une colonne de $\boldsymbol{H}$. Le clé de cette carte est l'indexe de la rangée et la donnée est une matrice 3×3. Posons $map[i, j]$ la j^e matrice de la i^e carte.

for all $e_{ij} \in \boldsymbol{e}$, supposons que $i < j$ **do**
  if c_i est vide **then**
    $map[i, i] += J_i^T \Lambda_{ij} J_i$
  end if
  if c_j est vide **then**
    $map[j, j] += J_j^T \Lambda_{ij} J_j$
  end if
  if c_i et c_j est vide **then**
    $map[j, i] += J_i^T \Lambda_{ij} J_j$
  end if
end for

Créer la matrice triangulaire supérieure éparse $\boldsymbol{H}$
On considère dans la suite seulement la partie triangulaire supérieure de $map[]$ et ignore la partie triangulaire inférieure
Aller dans $map[]$ dans l'ordre des colonnes puis des rangées pour chercher le nombre d'éléments non nuls dans chaque colonne (col_ptr) et leur rangée (row_ind)
Aller dans $map[]$ dans l'ordre des colonnes puis des rangées encore une fois pour insérer séquentiellement les données dans la matrice (val)

ANNEXE C ALGORITHME DE LEVENBERG-MARQUARDT AVEC λ PERSISTANT

Require:

$\boldsymbol{c}$, la liste des nœuds

$\boldsymbol{e}$, la liste des contraintes

λ, le facteur d'amortissement de l'algorithme LM

Ensure:

mise à jour $\boldsymbol{c}$

if λ =0 **then**

Mettre λ à l'ancien valeur

end if

Créer la matrice éparse $\boldsymbol{H}$ suivant l'annexe B avec $\boldsymbol{c_f} = \boldsymbol{c} - c_0$ avec c_0 la pose fixe

Résoudre équation 2.19 en utilisant la méthode de Cholesky

Mise à jour la variable $\boldsymbol{c} - c_0 + = \Delta \boldsymbol{x}$

if $\boldsymbol{e}$ diminue **then**

$\lambda = \lambda / 2$

Retour les poses mises à jour pour $\boldsymbol{c} - c_0$

end if

if $\boldsymbol{e}$ augmente **then**

$\lambda = \lambda * 2$

Retour les poses originales pour $\boldsymbol{c} - c_0$

end if

ANNEXE D ALGORITHME DE SÉPARATION ET ÉVALUATION DE L'ORIGINE

Require:
C_0, l'arbre origine
Ensure:
c^*, la solution optimale
S^*, la note maximale

$S^* = -\infty$
$C = C_0$
while $C \neq \emptyset$ **do**
  Évaluer un nœuds $c \in C$
  if c est une feuille **then**
    if $S(c) > S^*$ **then**
      $c^* = c$
      $S^* = S(c)$
    end if
  else
    if $S(c) > S^*$ **then**
      Séparation : Diviser c en ensemble des enfants C_c
      $C = C \cup C_0$
    else
      Éliminer les enfants de c dans C
    end if
  end if
  Éliminer c dans C
end while

ANNEXE E ALGORITHME DE SÉPARATION ET ÉVALUATION DE CARTOGRAPHER

Require:

C, l'arbre origine

C_0, les nœuds racines de C

S_c, La seuil du score

Ensure:

c^*, la solution optimale

S^*, la note maximale

$S^* = S_c$

Calculer et mémoriser la note des nœuds racines C_0.

Initialiser C avec les données de C_0 triées par le score, ordre descendant.

while $C \neq \emptyset$ **do**

  Pop c de C

  if $S(c) > S^*$ **then**

    if c est une feuille **then**

      $c^* = c$

      $S^* = S(c)$

    else

      Diviser c en ensemble des enfants C_c

      Calculer et mémoriser la note des éléments dans C_c.

      Pousser C_c dans la liste, trié par la note, les meilleurs sont les derniers.

    end if

  end if

end while